REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	2/15/97	Final

4. TITLE AND SUBTITLE

Creation of Efficient and Portable Parallel Programs

5. FUNDING NUMBERS

G N000149610800

6. AUTHOR(S)

Prof. Larry Wittie

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Research Foundation of
State University of New York
Computer Science, SUNY Stony Brook
Stony Brook, NY 11794-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

431-0619A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Math, Computer & Information Sciences Div.
Code 311/Dr. Andre M. van Tilborg, Director
800 North Quincy St.
Arlington, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This grant supported research to develop methods for parallel programming by masses of people using computers on networks. Scientific results have surpassed initial expectations. The convergence of little-known Russian developments in program transformation, supercompilation, and partial evaluation with the truly American phenomenon of Java programming have produced surprisingly strong results. This part of the pNet (programming parallel Networks) project is the pJava, or parallelizable Java, language. Particularly exciting is the development of program transformation techniques for machine understanding of parallel programs written in the familiar syntax of the important new Web programming language Java. The remainder of the research centers around the Norma language, developed to help applied mathematicians create efficient fluid-flow codes. Norma helps produce parallel programs that can run efficiently on large classes of parallel and distributed computers, including the Web.

The short eight months of research supported by this grant have produced many sound scientific results. This work points the way to a grand unification of techniques in program transformation, parallelization, and compilation that will allow the creation of libraries of reusable portable parallel codes that will run efficiently on almost any computer system to be found in or on the computer networks of the world.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Computer Languages, Portable Parallel Programs for the Web, Efficient Reusable Code, Java, Program Transformations, Supercompilation, Norma, Fluid-Flow Codes	190
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

Creation of Efficient and Portable Parallel Programs

Final report on grant ONR N000149610800 SUNY 431-0619A
Larry Wittie lw@cs.sunysb.edu 5/01/96-12/31/96 $84,824

The Principal Investigator made one trip to Moscow Russia in August 1996 and spent many hours on the internet in connection with this grant to develop methods for parallel programming by masses of people using computers and groups of computers on the network. Two Russian scientists visited New York for several weeks of collaboration and familiarization with American research facilities. These trips strengthened the scientific collaboration that had developed during eight prior trips to Russia by the Investigator to find Russian computer experts with technical results that could further world science if they were allowed to continue their research amid the gales of economic reform in Russia.

Scientific results have surpassed initial expectations. The Norma language, first developed to help applied mathematicians create efficient fluid-flow codes offers hope for producing parallel programs that can run efficiently on large classes of parallel and distributed computers, including the World Wide Web. Even more exciting is the development of program transformation techniques for machine understanding of parallel programs written in the familiar syntax of the important new programming language Java. Supercompilation techniques for very-high-level improvements of programs can work for Java programs, allowing masses of users to combine library routines in simple ways and yet produce highly efficient parallel codes.

The primary goals for the pNet project were: (1) To explore new principles, languages and methods for creating efficient portable parallel programs for massively parallel computers and networks of computers; and (2) To make it easy for technical application experts to specify elegant solutions to well understood, but possibly computationally complex information processing problems. The work on the Norma language has addressed these goals.

The secondary goals were: (1) To allow the creation and control of massively parallel(p) programs, run on possibly millions of cheaply available computers within huge geographically distributed networks(Net); (2) To simplify the creation of complex parallel programs using compositions of pretested program modules; (3) To make recent Russian developments for parallel computations available to scientists in western nations; and (4) To build lasting collegial and collaborative ties between American and Russian scientists.

The ONR sponsored series of personal scientific interactions between American scientists (including this Investigator) and computer scientists in the institutes of the Russian Academy of Sciences in Moscow and Novosibirsk have led to an unanticipated convergence of American and Russian computing cultures. Combining little-known Russian developments in the program transformation disciplines of supercompilation and partial evaluation with the truly American phenomenon of Java programming have produced surprisingly strong scientific results. The Russians taught the investigator about program improvements based on computer understanding and simplification of human produced programs. The Investigator suggested that the methods might speed up routinely interpreted Java programs without sacrificing the Java guarantees of safety

from rogue programs. In 1995, most Russian scientists had not then heard of Java. The result is a major part of the pNet project: that of developing the pJava, or parallelizable Java, language and program creation tools based on it. To produce these results, the Investigator and his Russian colleagues have had to share the chaos and privations of the Russian economic collapse, bridge long-instilled cold-war chasms of fear and distrust, and overcome an awkward language barrier. The results of the collaboration at at both the scientific and human levels have been worth the discomforts.

Much of the first six months of funding for this project were spent climbing the mountains of paperwork required to support five Russian researchers for eight months in 1996. The two-week trip to Moscow in August allowed the investigator to help build modern personal computer systems for research on Norma, pJava, and supercompilation at two research centers in Russia, Keldysh Institute of Applied Mathematics in Moscow and the Program Systems Institute in Pereslavl-Zalessky. Both institutes are part of the Russian Academy of Sciences. More importantly, it allowed his Russian colleagues to keep their hopes alive after waiting in vain four months to be paid for their work on the project. The trip also permitted the investigator to acquire identical dual English-Cyrillic keyboards for use in both Russia and the United States by team members. During their trip to the New York in November, his two visitors installed operating system software compatible with that on their machines in Russia. This seemingly small detail of having the same keyboard and compatible personal computer software has made possible the efficient production of joint scientific reports.

The bulk of the scientific developments from this project are contained in the six technical reports attached to this final report. Together they cover nearly two hundred pages of analyses in applied mathematics and computer science. Three of the reports are newly written for this project. Three are translations of Norma design documents previously available only in Russian. The rest of this report gives the abstracts of the six technical papers in the Appendix.

Program Transformations for Java

For the investigator, the most exciting work is given in the paper entitled, "Program Transformations for Java" by Andrei V. Klimov of the Keldysh Institute for Applied Mathematics in Moscow, Valentin F. Turchin of The City College of New York, and Larry D. Wittie of SUNY at Stony Brook in New York. It shows how supercompilation can be used to transform parallel and distributed programs written in (p)Java and make them much more efficient. It also gives clear examples of how a Java supercompiler works to understand the deep meaning of (p)Java programs and can produce simpler equivalent programs.

> A program transformation system for Java is presented. Two program understanding and transformation disciplines are reviewed: supercompilation and partial evaluation. Supercompilation is more general and powerful: partial evaluation is a subset, but simpler to understand and to use.
>
> Transformation methods were originally developed for functional languages. Here, for the first time, is a demonstration of supercompilation for the imperative language Java. We show how

the main phases of a supercompiler work for Java: configuration, driving, configuration analysis. How to supercompile Java is explained via a Producer-Consumer example, including a trace of manual supercompilation steps.

The strong points of this report are:
It gives an excellent review of program transformation methods;
It shows that the syntax of the popular Java programming language can be used to write computer-understandable portable parallel programs that automatically can be deeply analyzed, optimized and tailored to run efficiently for many specific problems or computer system architectures; and
Its major scientific contribution is a demonstration that supercompilation techniques are already powerful enough to fuse several concurrent threads into one, dramatically increasing the efficiency of some computer programs.

Supercompilers can perform significant inter-thread analysis and source code optimization. We explain why and how we plan to develop and implement a prototype supercompiler for Java.

Effectively a supercompiler can deduce the essential meaning of calls to many code functions and incorporate that meaning into highly-restructured faster-running codes.

pJava - A Parallel Superset of Java

The second report covers the planned steps in the development of an implicitly parallel version of the Java programming language, called pJava. The paper is entitled, "pJava - A Parallel Superset of Java for Automatic Parallelization" and was written by Andrei V. Klimov of the Keldysh Institute for Applied Mathematics in Moscow and Larry D. Wittie of SUNY at Stony Brook in New York.

An extension of Java, called pJava, which allows for automatic parallelization and efficient program transformation, is presented. The idea of pJava is to select a Java subset with suitable properties, and then gradually to extend it while preserving the properties. The starting point for pJava is a purely functional subset of Java with data limited to immutable objects. Higher-level declarative notions will gradually be added to allow for easy, reliable, implicitly parallel programming. Ultimately mutable objects will be allowed as well, but under a special programming discipline. This discipline can be satisfied at a low-level by a qualified Java programmer, or at a high-level via constructs based on monotone objects, which can be easily applied by less-experienced users.

The pJava language will allow programs to be written that can be ported to new computer systems and new application areas, and still be analyzed and improved automatically by supercompilation system algorithms. More rapid execution can result both from program transformations to produce faster, simpler, guaranteed-equivalent source codes and from automatic parallelization to use many computers at once on parts of the same code.

Creation of Efficient and Portable Parallel Programs

The third report is entitled, "Creation of Efficient and Portable Parallel Programs" and was written under the direction of Igor B. Zadykhailo of the Keldysh Institute for Applied Mathematics in Moscow. It gives a detailed semantic definition for the Norma parallel programming language for applied mathematics. This precise definition is needed to build the program-analysis systems that underlie both supercompilation transformations and automatic parallelization of mathematical solution algorithms for problems in applied mathematics.

> The first publications appeared in late 50s - early 60s. The main idea of the language later named NORMA is very simple. It was an attempt to automate the design of the programs based on the jobs prepared by applied mathematicians from Keldysh Institute of Applied Mathematics for further programming. Usually those jobs were the result of applying numerical methods (more often grid method) to physical problems' solution. The intention was to create the language of jobs' specification corresponding to the constructions obtained after mathematical solution of the problem.
>
> We have gained much experience in designing complicated program systems and translators, now it is time to create really friendly programming languages. We shall take a decisive step and turn from universal languages to the languages for users to formulate problem solution in generic terms. We are sure that universal languages may be friendly only to system programmer. Hence we shall bend every effort to creation of specialized language for each application domain.
>
> As the specification in the NORMA language contains full specification of an algorithm and doesn't reflect any peculiarities of the computer. It may be implemented on the computer with any architecture both sequential or parallel. Synthesising translator must be automatically adapted to the peculiarities of the architecture or must allow for the peculiar architecture.
>
> As the user is free from the necessity of making a program (this part is carried out by a translator) then such process of programming doesn't cause mistakes (up to the reliability of a translator). Further more besides the traditional syntactical and semantic diagnostics the synthesising translator can send messages about the errors in the essential notions. E.g., impossibility of organising computation caused by insufficient initial data. by the mistakes in the index displacements etc.
>
> High efficiency of automatically designed program is based on the capability of deep parallelising and providing the necessary level of parallelism granularity. Generally speaking the synthesising translator has the capability of estimating the different variants of the representation of the declarative specification in the program and finding of the best one according to the built-in rule or in the dialogue with the system programmer or the user himself.
>
> Formal semantics' specification of the NORMA language versions 1-22 are given in this part. Semantics' specification is based on the application of operational approach and formal methods of relational algebra. References to syntactical notations introduced in specification of the NORMA language by extended notation of Backus-Naur is used here.

A Norma to pJava translation system is among the first software products planned for this part of the pNet project. Not only do program analysis routines for Norma have much in common with those for pJava, but technical developments for pJava directly will make faster running Norma target codes for a wide range of computer architectures.

Translations of Recent Norma Papers From Russian

The last three reports are translations from Russian to English of recent papers in the development of the Norma programming language and its underlying program-analysis and -compilation systems. There are also four older Norma papers still only in Russian.

NORMA - Language specification - Draft copy 1.22

This is the latest version of the formal specifications of Norma language syntax, entitled: "NORMA - Language specification - Draft copy 1.22". It is a 1996 revision of the earlier 50 page preprint number120 "The specification of the NORMA language. Draft Standard," published at the Keldysh Institute of Applied Mathematics in Moscow in 1995 by the same authors: Alexander Nikolaevich Andrianov, Alexander Borisovich Bugerya, Kyrill Nikolaevich Efimkin, and Igor Borisovich Zadykhailo. It gives the form of all statements in the Norma language.

> The NORMA language is a tool aimed at automatic solution of the mathematical physics problems on parallel computer systems. The aim of the NORMA language is to eliminate the programming phase which is necessary to pass from computational formulae derived by an application specialist to a computer program. There is no essential difference between computational formulae and NORMA program structures - these formulae are an input for the NORMA translating system. In fact NORMA program is a nonprocedural specification of problems to be solved. The mathematical problems connected with the synthesis of output program are solvable in the case of the NORMA language. Draft specification of the NORMA language is given.

Organization of loop computations in NORMA language

This is an slightly older paper by Alexander N. (Sasha) Andrianov about the compilation methods to use in determining the execution order of operations in Norma loops: "Organization of Loop Computations in the NORMA Language." It was preprint number 171 printed in Moscow in 1986 by the Keldysh Institute of Applied Mathematics. The last of the three papers is a continuation of this report, which was 26 pages in Russian.

> The problems of loop process organization for the program written in nonprocedural language NORMA are considered in this paper. An algorithm of designing the system of simple loops allowing parallel processing is given. The algorithm is based on the notion of computation's front which is a hyperplane where variables' values may be computed in every its point. The task of Linear Integer Programming is solved for determination of the hyperplane's parameters.
> Key words: nonprocedural language. synthesis of the program. parallel computations.

Organization of loop process on specification

This paper "Organization of loop process on nonprocedural specification" by A.N. Andrianov and E.A. Andrianova is a continuation of the 1986 on organization of loop operations in Norma, number 2 in this list of translated papers. It was completed in Moscow in 1996.

NORMA is a programming language [1] aimed at automation of mathematical physics problems solutions on parallel computer systems.

The NORMA language allows elimination of a programming phase in transition from formulae specified by a technical expert to a program itself. There is no much difference between formulae and NORMA specifications. In fact these formulae are input data for a translator.

Synthesis of output program is carried out automatically during the translation from NORMA. The order and the way of performing calculations (parallel, vector or sequential) is determined automatically. The order of the language's sentences is arbitrary (information dependencies are revealed and taken into account during the organization of computing process). There are no such programming terms as memory, loop, control operators in the language. Output program is generated with the architecture of a target computer as a guide.

In fact the program in NORMA is a nonprocedural specification of the problem to be solved. The synthesis of output program raises some mathematical problems but they are solvable in the case of NORMA language.

Some Norma peculiarities makes the process of automatic object program design available for practice realization. They are:

1. Index expressions of calculated variables has the form $i \mp c$ where i - index name, c - integer constant.
2. NORMA is a language with single assignment. Any value can be assigned to a variable only once (only once to each point of domain - to the variables defined on domain).The first constraint defines the class of formulae which can be used for the problem's solution. It isn't strict in practice as the index expressions of other type are very rare.

Memory allocation and the problems of its economy caused the second constraint. These problems can be solved at the translation stage. The second constraint simplifies the problem of output program synthesis.

The problem of output program is to be solved during the translation. Solving this problem is based on the analysis of the graph of information dependencies. The Most Strongly Connected subGraphs (MSCG) are chosen from the graph. In general case organization of computations for the nodes requires use of special methods.

Assume is a principal operator in NORMA. This operator sets the relations between variables being calculated on a domain. Researching on the subject of computational process organization has been doing for a long time. The purpose of this paper is to specify the method of designing loop operators which realizes the relations included in MSCG (Most Strongly Connected subGraphs).

What we are discussing is an amalgamation of program understanding and transformation technologies to let computers, without help from ultra-skilled humans, write efficient working programs in the future world. Our computers are becoming powerful enough to understand, to manipulate and to improve their own programs, at least for most mundane applications. They are becoming complex enough that soon only

they will be able to write efficient codes for the many variants of parallel and distributed computing systems. Humans should only point the way to new algorithms for new problems. Computers should handle all details of producing fast running codes.

The short eight months of research supported by this grant have produced many sound scientific results. This work points the way to a grand unification of techniques in program transformation, parallelization, and compilation that will allow the creation of libraries of reusable portable parallel codes that will run efficiently on almost any computer system to be found in or on the computer networks of the world.

Appendix:

1)	**Program Transformations for Java**	**28 pp.**
2)	**pJava - A Parallel Superset of Java**	**10 pp.**
3)	**Creation of Efficient and Portable Parallel Programs**	**54 pp.**
4)	**NORMA - Language specification - Draft copy 1.22**	**46 pp.**
5)	**Organization of Loop Computations in the NORMA Language**	**25 pp.**
6)	**Organization of loop process on nonprocedural specification**	**19 pp.**

Program Transformations for Java

Andrei V. Klimov

Keldysh Institute for Applied Mathematics, Moscow

Valentin F. Turchin

The City College of New York

Larry D. Wittie

SUNY at Stony Brook, New York

Abstract

A program transformation system for Java is presented. Two disciplines are reviewed: supercompilation and partial evaluation. Supercompilation is more general and powerful; partial evaluation is a subset, but simpler to understand and to use. Transformation methods were originally developed for functional languages. Here, for the first time, is a demonstration of supercompilation for the imperative language Java. We show how the main phases of a supercompiler work for Java: configuration, driving, configuration analysis. Supercompilation of Java is explained via a Producer-Consumer example, including a trace of manual supercompilation steps.

The strong points of this report are:

It gives an excellent review of program transformation methods.

It shows that the syntax of the popular Java programming language can be used to write computer-understandable portable parallel programs that automatically can be deeply analyzed, optimized and tailored to run efficiently for many specific problems or computer system architectures; and

Its major scientific contribution is a demonstration that supercompilation techniques are already powerful enough to fuse several concurrent threads into one, dramatically increasing the efficiency of some computer programs.

Supercompilers can perform significant interthread analysis and source code optimization. We explain why and how we plan to develop and implement a prototype supercompiler for Java.

This research has been supported by the Office of Naval Research: grant 00014-96-1-0800.

Contents

Overview of Program Transformation Methods for Java

Introduction

Why supercompilation?

Among the variety of program transformation techniques, two methods perform especially deep optimization:

partial evaluation [Jones et al 93], and

supercompilation [Turchin 86, 88, 93, 96].

Originally, these were developed for pure functional languages, but now have become mature enough to be applied to wide-spread practical languages like Java. Partial evaluation is simpler than supercompilation and is better studied. The former is actually a case of the later. A year ago, in our proposal to ONR, we planned to start by constructing a partial evaluator for Java, and to turn to supercompilation only when we saw that the simpler method turned out well and we understood all subtleties in the transformation of Java programs. However, our research in 1996 has shown that we were too timorous. Hence, this report centers around supercompilation. Our main contribution is that we show how the problem of fusion of several threads into one can be solved by supercompilation. One of the reasons why we have laid partial evaluation aside is that it cannot fuse threads, although we do not see any significant problem in applying it to Java.

At the current stage of research, for Java we plan to use only well-established methods of supercompilation, ones which have already been proved for functional languages. Only after we see how these work in prototype implementation, will we consider using advanced techniques. This report presents the results of a redevelopment of basic supercompilation notions with respect to Java. We are pleased to find that the main ideas remain the same, although a lot of details are new.

Why Java?

Success in program transformation disciplines like supercompilation greatly depends on language properties. It is often said that functional languages are better than imperative for such a task. Indeed, partial evaluation applies to constant subexpressions, that is, program parts that belong to a functional subset. However, this is not essential for supercompilation, which models the operational behavior of a program. What actually matters for all kinds of program transformation is the *data* model of a language rather than *control*.

The majority of wide-spread languages, from Fortran to C++, are based on the data model of a von Neumann computer. This means that data definition is two-level. First, clear abstract data domains are defined: integers, strings, records, objects, etc. Second, these are mapped into bit sequences and the one-dimensional array of memory cells. A lot of operations, and even some of the data types (e.g. pointers), refer to the second part of the definition: data coercion and allocate/deallocate structures. This two-level data model badly complicates program analysis and transformation, and is a real (probably, the only) obstacle for bringing the achievements of computer science into practice.

The reason why functional and logic languages are better suited for advanced computing techniques is that from the very beginning they avoided the pressures of program efficiency, and accepted a clean one-level data model. The semantics of programs in languages with a one-level data model can be analyzed and manipulated more easily by other computer programs. Human programming skills are not needed for all aspects of the production of efficient code.

Only a few widely-known languages from the imperative family come close to having a good one-level data model: Algol-68, Ada, and Java. And that is all. We don't mention such languages as Oberon and Modula-3, since these were not widely used). Java is the simplest of the three, but the most interesting, since it actually supports two programming paradigms. It is both *object-oriented* and has a *functional* subset.

By saying that a language has a functional subset we imply that

- its data are developed well enough to represent complex values (at least trees) by immutable data structures;
- functions and procedures can return any values.

To satisfy these requirements, the language implementation must use *garbage collection*.

Experts in functional programming usually put forward a third requirement that higher-order programming is also supported, that is functions may be used as first class data. However, this is not important for program transformation. On the other hand, Java is good with respect to this feature as well, since functions as values are easily modeled by objects. Java satisfies the functional programming requirements best of all wide-spread imperative languages. Indeed, in [Wadler, Odersky 97] it is demonstrated that Java notions are sufficient to extend it syntactically to a fully functional programming language.

What is supercompilation?

This subsection gives an overview of the notions and algorithms of supercompilation. These are explained in more detail along with examples in the next section.

In short, a supercompiler executes a program in general terms, with some data replaced by *variables*, analyses the trace and constructs an equivalent *residual*[1] program, which is usually much more efficient that the original one.

In more detail, the definition of supercompilation is two-level. The lower level is a potentially infinite process called *driving*. The upper-level is *configuration analysis*, a process which *super*vises the trace of driving, performs various operations on the trace, controls driving, and constructs a finite graph of a residual program, thus, *compiles*. Hence, the term *super-compilation*.

The notion of a configuration

The central notion and the main object of the supercompiler is a *configuration*. A *configuration* is the representation of (a part of) a generalized state of the machine executing the source program. It resembles a state representation in an interpreter, but contains additional features. To represent sets, free variables, referred to as *configuration variables*, occur in configurations instead of ground values (final values). The details of Java configurations are defined in the next section.

Driving

The basic process in supercompilation is *driving*, that is execution of a source program in terms of configurations.

[1] In accordance with the partial evaluation tradition, we refer to a result program, its statements and expressions as *residual*. The act of generating a residual statement or a residual expression is called *residualization*.

Driving starts with one or more *initial configurations* supplied by the user. Each initial configuration generates a task to construct a residual procedure. It contains initial configuration variables, which become parameters of the residual procedure. In order just to *optimize* a procedure, the user gives the initial configuration that represents the call to the procedure with all arguments taking on different configuration variables. To *specialize* a procedure, the user gives known values to some arguments.

Driving gradually constructs a potentially infinite *driving tree* (also referred to as a *process tree*), which represents the set of all traces of source program execution for the set of its initial states covered by an initial configuration. The nodes of the tree correspond to configurations, the arcs to the steps of execution of the source program. The leaves are *passive* configurations that contain no procedure calls and usually correspond to the return statement yielding a result value.

A step of driving is either *transient*, or produces a *residual* statement *(a residualization step)*.

Transient driving is execution of steps as if by an interpreter, as long as configuration variables do not interfere. Transient steps result in a linear segment of the driving tree. No statements are residualized during transient driving.

When an unknown represented by a configuration variable prohibits execution of a source program statement, or a procedure called in the statement is not given, the statement is *residualized*. The values of program variables, including the initial configuration variables, are substituted into the statement, and the result statement is residualized and stored in the node of the driving tree together with the configuration. For an unconditional residual statement, only one arc proceeds from the node. Otherwise, two or more arcs leave the node. The arcs enter nodes corresponding to possible next configurations.

The next configuration may be *narrowed* to represent the information revealed by a step. For example, after a test "if x_3==5 ...", where x_3 is a configuration variable, it is known that x_3 equals to 5 on the positive branch. The narrowing information is represented by two means. First, by *contraction*, that is by substituting some values instead of configuration variables ($x_3 \mapsto 5$ in our example). Second, by adding the information to *restrictions*, which are kept together with configurations. In the basic case (which we plant to implement for Java), restrictions are inequalities (for example, $x_3 \neq 5$). More details are given in the next sections.

Configuration analysis

The driving tree may be considered as an infinite representation of the residual program. Indeed, define its interpretation as follows. Forget configurations, keep in nodes only residual statements and lists of configuration variables. Regard configuration variables as program variables. Assign values to the initial configuration variables and start from the initial node. Execute the residual statement kept in the current node. If only one arc starts from the node, go to the next node. If several arcs proceed from the current node, then the residual statement is conditional, and its execution has selected one of the arcs. Go to the corresponding node. And so forth, until a leaf is reached. The value returned by the expression in the leaf is the result of interpretation.

The interpreter would work similarly if given a finite graph rather than a tree. The finite graph would be an eligible result of compilation. So, the only (but the hardest!) problem is to fold the infinite driving tree into a finite graph. This is the task for configuration analysis.

The following operations on the tree can be performed to fold it:

Looping back: If a configuration C is a subset of one of the previous ones, C', and can be reduced to it by a substitution μ, $C = \mu C'$, a looping back arc (corresponding to a goto statement in the residual program) is added to the graph plus appropriate assignment statements reflecting the substitution μ.

Generalization: A subtree starting from a node with configuration C can be thrown away and replaced by a subgraph starting with a more general configuration C^g, such that $C = \mu\, C^g$ for some substitution μ. The residual assignments corresponding to the substitution μ are put into node C. The new graph represents an equivalent but less efficient residual program.

Cutting a configuration: A subtree starting from a node with configuration C can be thrown away and replaced by two subgraphs starting with configurations C_1 and C_2, which are parts of the original configuration C, and C is a kind of composition of C_1 and C_2. We will not go into details yet. However, the residual statement corresponding to C is a call to a procedure C_1, while C_2 is the continuation of C after C_1 returns.

These operations are sufficient to fold any infinite driving tree into a finite graph. However, to detect particular configurations to loop back, to generalize or to cut is the hardest problem of supercompilation. As a first approximation, we will use the method described in [Turchin 88] after modifying it for multiple threads. The idea is that only the structure of threads is used to make decisions. The values bound to program variables are not taken into account. This method guarantees termination of supercompilation. The details are beyond the scope of this report.

What is partial evaluation?

Partial evaluation is a simplification of supercompilation in several respects.

1. In supercompilation one residual program points can emerge from several source program points. This property is referred to as *polygenetic*, versus *monogenetic* [Romanenko 90]. Partial evaluation processes each procedure separately. Hence, one residual program point corresponds to only one source program point. This is *monogenetic* program transformation. Several residual procedures can still be produced from one source procedure. In partial evaluation, there is no problem in determining how to cut a configuration. We may say that cutting happens immediately at each procedure call.
2. A partial evaluator executes subexpressions depending only on known values. Cases when unknowns are present in data, but do not interfere, are not considered as executable by partial evaluation and are always residualized. Compared to supercompilation, partial evaluation leaves many more statements for execution, not eliminated at compile time.
3. The decision when and what to generalize is taken in advance, before known arguments are supplied, by a preprocessor that performs a so-called *binding time analysis* (BTA). The problem of generalization is drastically simplified as well for partial evaluation.

Binding time analysis

Partial evaluation is used to *specialize* procedures. Given a procedure of several arguments, say void P(int x,y,z), one wants to produce a procedure Q with fewer arguments, say z, with others having been bound to known values, say x=1, y=2. The implicit definition of Q is:

```
void Q(int z) {
    P(1,2,z);
}
void P(int x,y,z) {
    . . .
}
```

A partial evaluator generates specialized versions of codes by executing *static* subexpressions, ones that depend only on parameters with known values. All other expressions are *dynamic*. Before the

source program is simplified, it is evaluated by *binding time analysis* (BTA) to separate and annotate static subexpressions. A parameter of a procedure is considered static if it takes static arguments in all procedure calls. BTA iterates its *abstract interpretation* until a fixed point is reached. When it starts, all parameters of all procedures are hypothetically considered *static*, except the *dynamic* parameters of the main procedure. Information about being dynamic is propagated along the functional dependencies of the procedures. Whenever an argument in a procedure call is changed to dynamic, the corresponding parameter becomes dynamic as well, and the procedure body is reanalyzed.

In the P:Q example, if procedure P calls itself recursively in such a way that the second argument y depends on the third z, and the first argument x does not depend on y or on z, x is static and y and z are dynamic.

Partial evaluation proper

The second phase is the partial evaluation proper, that is the generation of specialized procedures when known values are given. Static parameters of a procedure may take several different values during this process. Respectively, several specialized versions of the same procedure are generated.

The only requirement is that the number of different values of each parameter must be finite. If this does not hold, the process does not finish, the user should stop it after waiting for some time, and study the dump to find an argument that takes infinite number of values, and forcefully declare it dynamic. Then BTA and partial evaluation are repeated.

Partial evaluation as a case of supercompilation

The role of the distinction between static and dynamic parameters can be explained in terms of generalization in supercompilation as follows:

- static arguments are *never* generalized;
- dynamic arguments are *always* generalized to a configuration variable.

The user can turn static parameters or subexpressions to dynamic by hand, thus controlling generalization. Simplicity is the main benefit and the main limitation of partial evaluation.

Benefits and limitations of partial evaluation

If something can be done by a simpler method, it should be done by it. A great discovery by the authors of partial evaluation was that it is sufficient to convert interpreters to compilers, solving a large class of practical problems. The main benefit of partial evaluation is that it allows clear control by a user who is developing an interpreter and allows experiments with specializing it:

- BTA results are simple for a user to read and to understand. They are just comments annotating the source program. A user may increase the level of generalization (residualization) by marking known (static) variables and subexpressions as unknown (dynamic).
- Termination properties are clear to a user: whether evaluation of static subexpressions results in a finite or infinite set of values of procedure arguments.
- Termination is easily controlled by a user: if a procedure argument takes on an infinite set of values, the user should forcefully mark it as dynamic.

The main limitations of partial evaluation are:

- It does not try to generalize complex configurations than BTA did not pick in advance.

- It processes procedures and threads separately and cannot fuse them (monogenetic).
- It evaluates only subexpressions belonging to a functional subset.

Composite system of partial evaluation and supercompilation

We have two powerful program transformation methods which can be implemented for Java:

- One is more general, powerful and complex: supercompilation;
- The other is less powerful, but simpler to understand and to use: partial evaluation.

Supercompilation is not a just a single method. It is a series of methods, each based on previous ones. However, in the nearest future, we plan to implement a prototype of just the *basic* supercompiler, discussed in this report. A practical program transformation system should contain program transformers of different power and with different features. It would be advantageous to have both partial evaluation and supercompilation for Java.

In such a dual system, the two program transformers should be used in the following order:

1. A program should be optimized by the simpler one, partial evaluation;
2. The supercompiler should be applied to the result of partial evaluation.

The supercompiler can perform the work of partial evaluator as well, but the later can do this more efficiently, especially when the subject program is still under development and its author needs to perform a series of experiments to understand how in behaves under program transformation. Having two program transformers for one practical language, we'll have a unique opportunity to compare these and get invaluable experience in using different techniques for constructing libraries of reusable software.

However, we plan first to concentrate on basic supercompilation. If we have support enough, a Java partial evaluator will be developed by another researcher.

Configuration Representation

A *configuration* is a generalized state of a machine executing a source program. It resembles a program state representation in a Java interpreter, but contains *configuration variables*.

We graphically denote configurations as follows.

Figure 1

Threads:

"thread name"	☺ *or* ☹
procedure name	program point
variable name	variable value
...	...
procedure name	program point
variable name	variable value
...	...
...	

...

Objects:

class name	🗀 *or* 🗁
variable name	variable value
...	...
(sub)class name	
variable name	variable value
...	...
...	...
(sub)class name	
variable name	variable value
...	...
...	
configuration variable (optional)	

...

class name	🗀 *or* 🗁
admission queue	list of ref. to threads
wait queue	list of ref. to threads
variable name	variable value
...	...
(sub)class name	
variable name	variable value
...	...
...	...
...	

Object with synchronized methods

Restrictions: set of inequalities

More formally, and in more detail, the representation of a configuration is defined as follows:

configuration	=	(set of threads, set of objects, set of restrictions)
thread	=	(thread name, active ☺ *or* passive ☹, list of stack frames)
stack frame	=	(procedure name, program point, list of variable bindings)
object	=	(list of (class name, list of variable bindings) optionally ended by a configuration variable, safe 🗀 *or* unsafe 🗁)
variable binding	=	(variable name, variable value)
variable value	=	ground value *or* configuration variable
ground value	=	simple type value *or* reference to an object
simple type value	=	integer *or* floating point number *or* character *or* Boolean

Configuration

A *configuration* is a triple of a set[2] of threads, a set of objects and a set of restrictions.

Threads and objects are identified by *references*. References to threads can be assigned to variables of the built-in class Thread and to an admission queue and a wait queue that are special variables automatically added to an object with synchronized methods. References to objects are used as variable values. We denote a reference by a dot with an arrow going to the respective object or thread. We have no reason to treat arrays separately as it is done in the Java specification, and interpret these like instances of a built-in class, say Array, with the obvious definition.

Threads

A *thread* consists of a thread name, a Boolean tag, ☺ or ☹ marking whether the tread is active or passive, and a stack of procedure calls (a list of stack frames). The *thread name* is a string passed as an argument to the Thread constructor. We use it as a comment to clarify the example (e.g. "mainThread", "Producer", "Consumer"). A thread is *passive* if the reference to it occurs in an admission or wait queue. Otherwise it is *active*. Java semantics allow a thread to wait only in one queue. An *admission queue* lists the threads waiting to enter a synchronized method. The threads that have executed wait() and have not received notify()stand in a *wait queue*.

A *stack frame* consists of a procedure name, a program point and a list of bindings that are pairs of a variable name and a variable value. In our examples, the program point is a number or a letter E that can be found in the program text to the right of the statement; it denotes a point that has been just executed. If a procedure is not static, then the first binding is for the variable this. That is, we consider a method call of form object.method(args)as a procedure call of form class.method(object.args), where the method in defined in the class.

Objects

The main contents of an *object* are variable names and values. A real supercompiler may avoid spending memory for variable names by renaming variables to ordinal numbers. These are organized in a list of subclasses: first, the top class name and its variables, second, the class name of a first inheritor and the variables defined in it, then the second inheritor, and so on. The list of subclasses may be ended by a configuration variable (see below).

An object has an attribute, *safe* or *unsafe*, denoted in figures as 🗀 and 🗁. Its use is as follows. A configuration represents part of a run-time residual program state. Other runtime threads may refer to the objects of a current configuration and may change them as well. Hence, a supercompiler must be able to imply that variable values may change at any moment, that their object is *unsafe*. However, if it knows that an object is referenced only from the current configuration, it know that its values change only explicitly during driving. Such an object is marked in a configuration as *safe*. The *safe* attribute is present only at (super)compile time and takes no space at run-time. In our examples all objects are safe. See the definition of driving for details of keeping the *unsafe* attribute.

[2] A set is an unordered finite list. Of course, in a computer, sets are represented by lists with some fixed order. By saying "set" we imply that its order does not matter as well as that we must check different orders of corresponding lists while comparing configurations during supercompilation.

Configuration variables

The supercompiler computes with partially unknown data. The substitutes for unknowns are referred to as *configuration variables*. A configuration variable may occur instead of any variable value in threads and objects. We denote configuration variables by italic identifiers with subscripts: n_1, q_3 etc. For clarity in examples, we use the same identifier for a configuration variable as the name of the program variable, which supplies the initial value for that configuration variable. A configuration variable may be copied and assigned to any thread or object variable, and may occur in several places. This represents information about the equality of values in different places. If a configuration variable remains instead of a ground value whena value must be known to perform an operation (e.g. in the if construct), an appropriate statement is put into the residual program. The new configuration may be either the same as before, or changed. In particular, a configuration variable may be *contracted*, that is replaced by an expression which represents information that has become known. See the definition of driving for details.

One more case of an unknown in a Java configuration is the tail of a list of subclasses in the object representation. When an object has been passed to a procedure as a parameter, say x, of a not final class C, it may actually belong either to C, or to some subclass C of C'. This is not known until x is tested by x.isC'() (or some other Java method). Before the test, the uncertainty is represented by a *tail configuration variable*. After the test, the configuration variable is *contracted* and (on the positive branch) replaced by a structure corresponding to the subclass C', and a new tail configuration variable if C' is not final as well.

Restrictions

Information about configuration variables is propagated by the supercompiler by two means. First, by *substitution* to reflect an equality: a configuration variable may be replaced by a ground value or another configuration variable. This operation is referred to as contraction of the variable. This happens when it becomes known that a variable equals to a constant or another configuration variable. In the special case of a tail variable, the replacement is a structure as just described.

The opposite case of inequality is impossible (or rather, inconvenient) to represent by a substitution. Inequalities are collected as *restrictions* to the set of states represented by a configuration.

In principle, the supercompiler can work and produce meaningful results without keeping restrictions. The result would be less accurate, the residual program would be larger and less efficient, but correct. It is the essence of supercompilation that the ideal result cannot in general be achieved, and from time to time a supercompiler must forget some restrictions, or perform some other kind of *generalization*, in order to construct a finite residual program.

The class of restrictions to keep with configurations and the algorithm to compare configurations while accounting for restrictions are a matter of heuristic choice by a supercompiler designer. Keeping in mind a particular class of tasks, we plan to implement the following restrictions in the Java supercompiler:

variable	$<$	constant	variable	$<$	variable
variable	$\leq$	constant	variable	$\leq$	variable
variable	$=$	constant	variable	$=$	variable
variable	$\neq$	class name			

Producer-Consumer Example

Introduction

Threads are used in imperative languages like Java not only for load-balancing multiprocessor systems, but for well-structured programs in object oriented paradigms. However, a highly structured program is often less efficient on a monoprocessor or on a system where the number of processors is significantly less than the number of threads in the program. The following example demonstrates that supercompilation can fuse threads and, in particular, transform a well-structured multithreaded program into a single-threaded efficient one.

Example

Consider a traditional example of a consumer and a producer communicating via a queue object. The example is based on one from the "Java Handbook" by Patrick Naughton, Osborn McGraw-Hill, pp.191-194. The program below consists of a class Queue that implements a queue of integers, a class Producer, a class Consumer[3], and a procedure doit that constructs Queue, Producer and Consumer instances. Two new threads start in Producer and Consumer.

Class Queue

The class Queue has 2 operations:

```
void put(int n);
int get();
```

The first operation puts a integer into a queue. The second operation gets the integer from the queue. The queue can hold at most one element.

```
class Queue {
    int n;
    boolean valueSet = false;
    synchronized void put(int n) {    0
        if (valueSet)                 1
            try wait(); catch(InterruptedException e);                2
        this.n = n;                   3
        valueSet = true;              4
        notify();                     E
    }
    synchronized int get() {          0
        if (!valueSet)                1
            try wait(); catch(InterruptedException e);                2
        valueSet = false;             3
        notify();                     4
        return n;                     E
    }
}
```

[3] The class Queue is a copy of the class Q on p.194 in all but its name. The classes Producer and Consumer differ slightly from that on p.192 in the procedure run.

This is a general purpose class which may be thought of as belonging to a library. The following two classes are simple examples of a producer and a consumer, which may be thought of as being written by an end user.

Class Producer

A Producer reads k integers by calling a function My.readInt, and puts them into the Queue supplied by the parameter q. It starts a separate thread.

```
class Producer implements Runnable {
    Queue q;
    int k;
    Producer(Queue q, int k) {          0
        this.q = q;                     1
        this.k = k;                     2
        new Thread(this, "Producer").start();                    E
    }
    public void run() throws IOException {                       0
        for (int i=1;                   1
                i<=k; i++) {2
            int n = My.readInt();  3
            q.put(n);                   4
        }                                                        E
    }
}
```

Class Consumer

A Consumer takes k integers from the Queue supplied by the parameter q, applies a function My.F to them and outputs the result by calling a function My.writeInt (the class My and the functions My.readInt, My.F, and My.writeInt are not specified at supercompile time). It starts a separate thread.

```
class Consumer implements Runnable {
    Queue q;
    int k;
    Consumer(Queue q, int k) {          0
        this.q = q;                     1
        this.k = k;                     2
        new Thread(this, "Consumer").start();                    E
    }
    public void run() {                 0
        for (int i=1;                   1
                i<=k; i++) {2
            int n = q.get();        3
            int m = My.F(n);            4
            My.writeInt(m);             5
        }                                                        E
    }
}
```

The class ProduceConsume

The task for the supercompiler is to optimize a procedure doit in a class ProduceConsume, which demonstrates a particular use of the Queue, Producer and Consumer classes. It supplies the number of integers to produce and consume, k, to Producer and Consumer instances . Below, 3 versions of the doit procedure and respective expected results of supercompilation (computed manually keeping in mind a particular supercompilation strategy) are presented.

The 1st and 2nd versions are substantionally the same case: these differ only in the order of the Producer and Consumer are constructed and the respective threads are started. Implying that the supercompiler uses a particular order of evaluation (*driving*) of threads (we use "elder threads are evaluated earlier"), the residual programs differ in the order of calls to My.readInt and My.writeInt. The essence of these versions is that the loop parameter is *static*, k=4. Hence, the residual program has simple structure: it is linear. The main problem supercompiler must solve here is not to stop too early. Otherwise, the threads would not be fully fused into one as in the expected results of supercompilation shown below. The criteria to continue driving that has been used during manual supercompilation is natural: the loops are controlled by inequalities of form i<=k, which involve no configuration variables, and the difference between the compared integers decreases.

The 3rd version differs essentially: the loop parameter k is *dynamic*. Hence, the residual program contains a loop on k (the criteria above does not work, since the loop conditions have configuration variables). The main problem supercompiler must solve here is to catch a moment when to construct a *basic* configuration, that is a point in the residual program to loop back. The algorithm to estimate 2 configurations as being similar and to *generalize* them is based on the paper [Turchin 88]. The idea is that to estimate similarity only the form of thread stacks is compared, while the values of program variables are ignored. However, all values are used to construct the least general configuration. Manual supercompilation of the 3rd version is shown below.

1st version: Static loop parameter; Producer starts before Consumer

Class ProduceConsume

```
class ProduceConsume {
    public static void doit() {        0
        int k = 4;                     1
        Queue q = new Queue();         2
        new Producer(q, k);            3
        new Consumer(q, k);            E
    }
}
```

Expected result of supercompilation

```
class ProduceConsume {
    public static void doit() {
        int n1 = My.readInt();
        int n2 = My.readInt();
        int n3 = My.readInt();
        int m1 = My.F(n1);
        My.writeInt(m1);
        int n4 = My.readInt();
        int m2 = My.F(n2);
        My.writeInt(m2);
        int m3 = My.F(n3);
        My.writeInt(m3);
        int m4 = My.F(n3);
        My.writeInt(m4);
    }
}
```

2nd version: Static loop parameter; Consumer starts before Producer

Class ProduceConsume

```
class ProduceConsume {
    public static void doit() {        0
        int k = 4;                     1
        Queue q = new Queue();         2
        new Consumer(q, k);            3
        new Producer(q, k);            E
}   }
```

Expected result of supercompilation

```
class ProduceConsume {
    public static void doit() {
        int n1 = My.readInt();
        int m1 = My.F(n1);
        My.writeInt(m1);
        int n2 = My.readInt();
        int m2 = My.F(n2);
        My.writeInt(m2);
        int n3 = My.readInt();
        int m3 = My.F(n3);
        My.writeInt(m3);
        int n4 = My.readInt();
        int m4 = My.F(n4);
        My.writeInt(m4);
}   }
```

3rd version: Dynamic loop parameter; Consumer starts before Producer

Class ProduceConsume

```
class ProduceConsume {
    public static void doit(int k) {    0
        Queue q = new Queue();          1
        new Consumer(q, k);             2
        new Producer(q, k);             E
}   }
```

Expected result of supercompilation

```
class ProduceConsume {
    public static void doit(int k) {
        for (int i=1; i<=k; i++) {
            int n1 = My.readInt();
            int m1 = My.F(n1);
            My.writeInt(m1);
}   }   }
```

Two interacting threads have been replaced by a single thread. Supercompilation has found one efficient loop that is equivalent to the entire interaction!

Producer-Consumer Example: Supercompilation Trace

In this section the main steps of the supercompilation process for the Producer-Consumer example are presented. See previous section for the text of the Java program. We consider the 3rd version of the example: dynamic loop parameter; Consumer starts before Producer. Here is the reminder of the procedure to be supercompiled:

Class ProduceConsume

```
class ProduceConsume {
    public static void doit(int k) {    0
        Queue q = new Queue();          1
        new Consumer(q, k);             2
        new Producer(q, k);             E
    }
}
```

Initial configuration

Supercompiler starts with one (or more) initial configuration(s). If we are interested in optimizing some procedure, the initial configuration represents the call to it with all parameters bound to distinct configuration variables. In our case, this is the call ProduceConsume.doit(k_1), where k_1 is a configuration variable that stands for an unknown value of the parameter k.

Figure 2

Threads:

"mainThread"	☺
ProduceConsume.doit	0
k	k_1

Objects: none

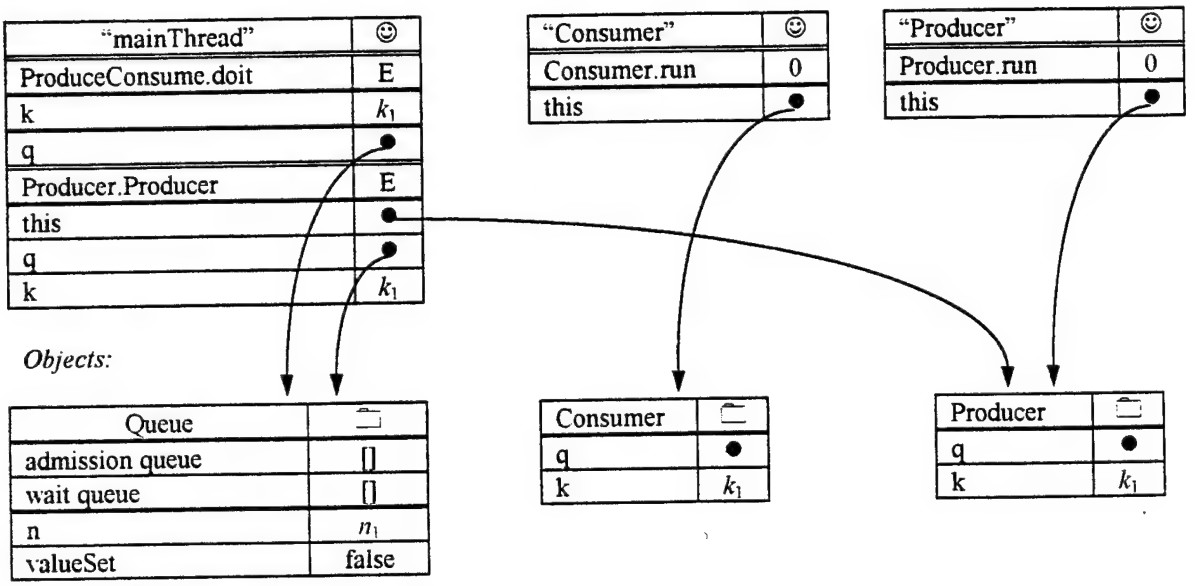
Restrictions: none

Configuration at the end of the Producer constructor

While configuration variables do not interfere, the supercompiler performs *transient driving* steps that execute a program like a common interpreter. For Java, transient driving imitates execution of a generally multithreaded program on a single processor. Some particular order to evaluate threads must be chosen. Any order is acceptable, but the residual program may differ depending on the order. This example uses the order "eldest thread executes first". When the next step of the eldest thread cannot be executed, either because it has come to a passive state, or because a configuration variable needs a value used to select one branch of a conditional statement, the next younger thread executes, and so on. After transient driving of all threads has completed, a branching in the residual program is generated, and the process goes further along one branch, then another, as needed. Branching details are discussed below. First, we will see how the supercompiler performs its initial transient driving steps, which are equivalent to the steps of an interpreter.

Figure 3 shows the configuration when mainThread has come to the end of the procedure Producer.Producer at program point E, as marked to the right of the program text. The mainThread dies after procedures Producer.Producer and ConsumeProduce.doit return. The threads Consumer and Producer have just been created and are standing at the beginning of their respective run() procedures.

Figure 3



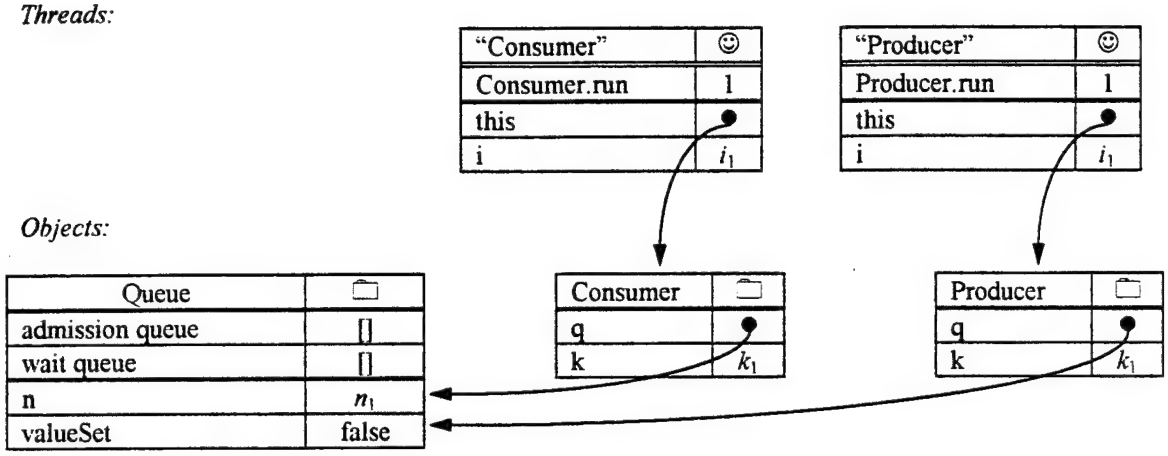
Restrictions: none

After mainThread dies, the newly eldest thread Consumer starts executing and enters its loop. Its integer variable i is initialized to 1 and the loop condition i<=k must be checked. After substituting the values i=1 and k=k_1, the supercompiler tries to evaluate 1<=k_1 and notices that configuration variable k_1 does not allow a branch to be choosen, so transient driving stops for thread Consumer. Thread Producer starts executing and stops in a similar state when it tries to check i<=k, as 1<=k_1, at the beginning of its loop. The two threads are evaluating different procedures, Consumer.run and Producer.run, and are checking different i and k variables, which just coincidentally are called by the same local names. Their full names are Consumer.run.i and Producer.run.i, Consumer.run.this.k and Producer.run.this.k. Coincidentally, the is and ks have the same values. 1 and k_1 respectively. The supercompiler will notice these coincidences and generate efficient code.

Basic configuration: Consumer and Producer to check loop condition i <= k

Figure 4

Threads:



Restrictions: none

Residual program constructed till now:

```
        public static void doit(int k) {
//      Initial configuration.
            i1 = 1;
//      Basic configuration.
        M1:
//      ... (Dots denote the residual program is not yet constructed here).
        }
```

The configuration in Figure 4 corresponds to the start of the loop in the residual program. Configurations that have two or more predecessors in the residual program graph are called *basic*. They mark the start of new basic blocks, sequential code sequences amenable to simple transient driving and points where variables used in two or more control threads must be made compatible. Constructing basic configurations is the main task of a supercompiler. It is the most complex problem in supercompilation. Basic configurations emerge as a result of loops.

In this example, when the supercompiler first comes to this point, there is no configuration variable i_1. The number 1 is the value for each local *i* variable. The supercompiler continues (details of how this happens are shown below), executes the first iteration of the loops in Consumer and Producer, and comes to a similar configuration, but with constant 2 for each *i*. The supercompiler must stop iterating and *generalize* two similar configurations. The heart of the problem for generating residual program loops is recognizing similar configurations.

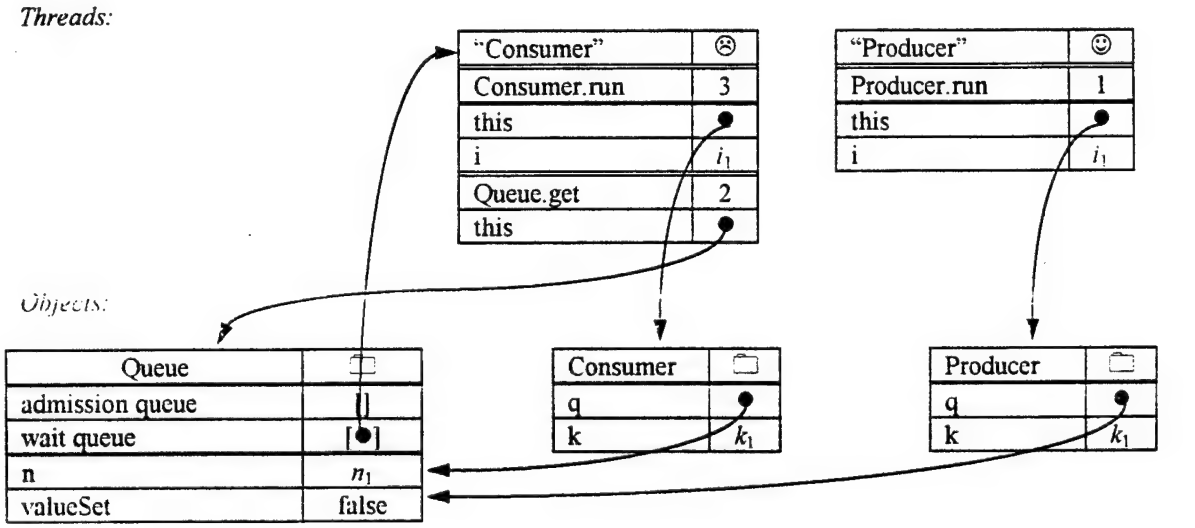
Deciding whether to stop driving after having met similar configurations is the most complex problem of supercompilation. The basic method shown in this example and planned for our Java implementation comes from the paper [Turchin 88]. This method of comparing and generalizing configurations was developed for lazy evaluation rather than for multiple threads run in applicative order. However, with small changes it can also be used for Java threads.

We do not discuss the general case here, just an idea of how it works for this example. Configurations with stack frames that have the same structure but different variable values are considered similar. When the current configuration is similar to a previous one, the supercompiler stops, and constructs *the least common generalization* of the configurations. It throws away the parts of the residual program graph that start with the old configuration, appends the generalized configuration the remaining program graph, and puts an assignment of new configuration variables just before the new generalized configuration.

Figure 4 shows the least common generalization of two similar configurations which differed in two places where the first had constant 1 instead of variable i_1, the current had constant 2. The assignment i1 = 1 is added to the residual program in generalizing the old configuration.

Configuration when Consumer waiting and Producer at start of loop body

Figure 5



Restrictions: $\{i_1 \le k_1\}$

Residual program:

```
        public static void doit(int k) {
//      Initial configuration.
            i1 = 1;
//      Basic configuration.
        M1: if (!(i1<=k1)) goto M2;
//      Current configuration: like the basic one but with restriction {i1 ≤ k1}.
//      ...
//      Configuration like the basic one but with restriction {i1 > k1}.
        M2:
//      ...
        }
```

Between the basic configuration in Figure 4 and the configuration in Figure 5, the supercompiler reasons as follows. The next step after the basic configuration cannot be executed unambiguously because of configuration variables in the condition $i_1 \leq k_1$. In such a (general) case, *driving* is performed as follows. The if statement with an unevaluated condition is put into the residual program, and the process of supercompilation continues along one of the branches. When, sooner or later, it has finished, the supercompiler will continue driving of the second branch. In our example, the if statement in the residual program looks as follows:[4]

```
    if (!(i1<=k1)) goto M2;
    // Positive branch for condition  i1 ≤ k1
M2: // Negative branch
```

At best, configurations on the branches should represent the exact sets of states into which a conditional statement has split a configuration. At worst, driving of each branch may continue from the same configuration. This is only a question of the quality of the residual program. There is a general rule of supercompilation: "At any moment, a configuration may be replaced by a more general one, reducing the quality of the residual program, but preserving correctness."

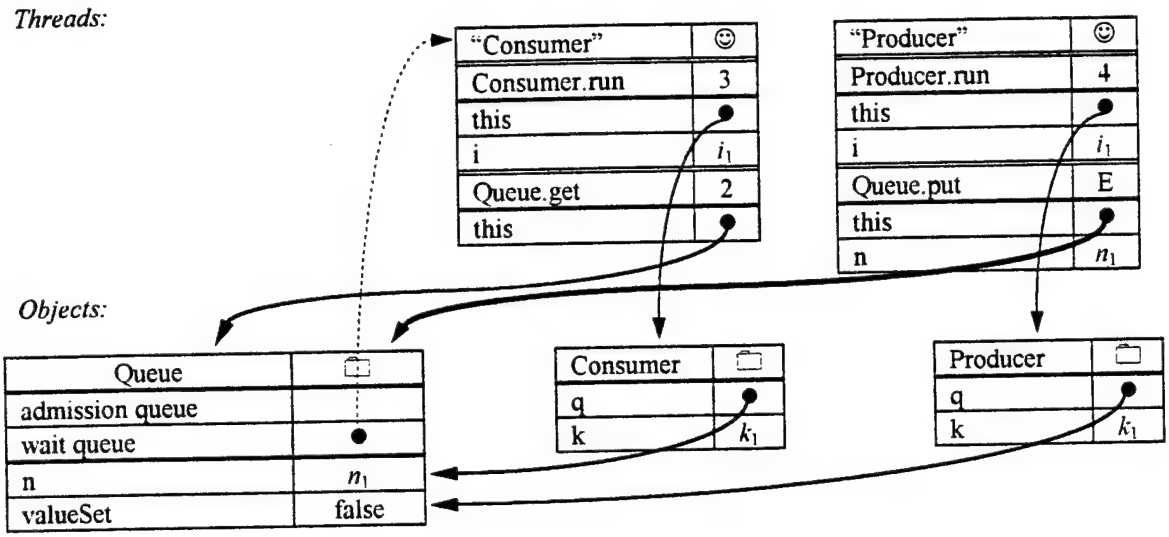
The planned implementation of a supercompiler for Java will be able to represent the exact splitting of the configuration in this example. The restriction $\{i_1 \leq k_1\}$ is added to the configuration on the positive branch, and $\{i_1 > k_1\}$ on the negative.

The supercompiler continues driving the positive branch and executes the Consumer thread until it has called wait() in the procedure Queue.run and has been put into the wait queue. This moment is shown in Figure 5.

[4] Although Java has no goto statement, we use it, since it is a natural means to represent an arbitrary graph of a residual program as a plain text. We'll use a postprocessor to translate gotos to legitimate Java constructs. (See discussion below).

Configuration when Producer notifies

Figure 6



Restrictions: $\{i_1 \leq k_1\}$

Residual program:

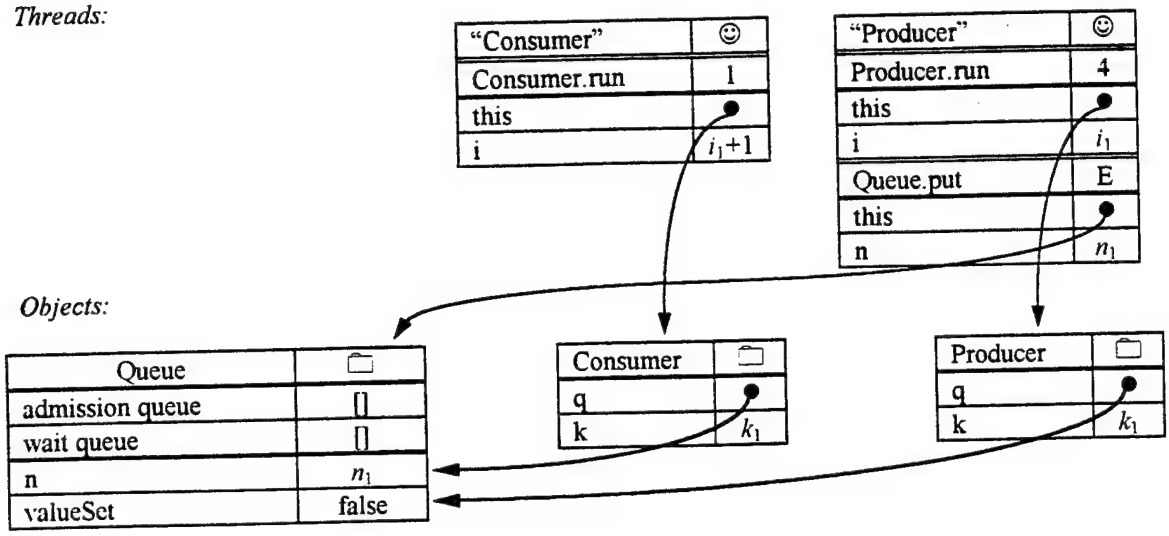
```
        public static void doit(int k) {
//      Initial configuration.
            i1 = 1;
//      Basic configuration.
        M1: if (!(i1<=k1)) goto M2;
//      Configuration like the basic one but with restriction {i1 ≤ k1}.
            int n1 = My.readInt();
//      Current configuration.
//      ...
//      Configuration like the basic one but with restriction {i1 > k1}.
        M2:
//      ...
        }
```

While the Consumer thread is passive, the Producer thread executes. The supercompiler behaves like an interpreter until it meets the statement int n = My.readInt(). The procedure My.readInt is unknown at supercompile time, and we have nothing to do but to represent the value of n by a new configuration variable n_1, and to generate the statement int n1 = My.readInt().

Then the Producer goes further and stops when it calls notify() and releases the Consumer. The current configuration is shown in Figure 6.

Configuration when Consumer to check loop condition for the second time

Figure 7



Restrictions: $\{i_1 \le k_1\}$

Residual program:

```
        public static void doit(int k) {
//      Initial configuration.
            i1 = 1;
//      Basic configuration.
        M1: if (!(i1<=k1)) goto M2;
                int n1 = My.readInt();
                int m1 = My.F(n1);
                My.writeInt(m1);
//      Current configuration.
//      ...
//      Configuration like the basic one but with restriction {i1 > k1}.
        M2:
//      ...
        }
```

After the Consumer thread has been released by notify(), two statements calling the procedures My.F and My.writeInt are added to the residual program, since the procedures are unknown. A new configuration variable m_1 represents the unknown value of program variable m.

At the end of the Consumer's loop, the variable i = i_1 must be incremented by 1. The supercompiler can behave in different ways. At simplest, it applies the general rule: when a statement or an expression cannot be evaluated, a new configuration variable (say i_2) is used to represent its result and an assignment to it is added to the residual program:

```
int i2 = i1+1;
```

Then i_2 becomes the value of the program variable Consumer.run.i.

A smarter supercompiler can postpone generating the residual statement and represent the expression value by a symbolic expression i_1+1. The class of expressions to be manipulated by the supercompiler

must be chosen by the designer of supercompiler for a particular language and data types. We plan to take this decision for Java after a series of experiments. For integers, it seems reasonable to support linear expressions, $a\,x + b$, where a and b are integer constants and x is a configuration variable.

Figure 7 shows the current configuration for the second method: the value of Consumer.run.i is i_1+1.

The supercompiler continues transient driving of the thread Consumer, reaches the beginning of the loop for the second time, and stops, since the loop condition i<=k evaluates to $i_1+1 \le k_1$, which involves configuration variables. The current configuration does not change and is still as in Figure 7.

Configuration when Consumer and Producer start their loops again

After the Consumer thread has been suspended by configuration variables, the supercompiler switches to the Producer thread and drives it until it comes to the beginning of the loop. On the way, the program variable Producer.run.i has been incremented and has taken on the value i_1+1 just as explained for Consumer.run.i .

When both loops restart, the current configuration looks like the basic one in Figure 4 except for the values of two program variables, Consumer.run.i and Producer.run.i, and restrictions. The values of both are i_1 in the basic configuration and i_1+1 in the current configuration. There is no restriction in the basic configuration, and $\{i_1 \le k_1\}$ in the current one. To loop a current configuration back to a basic one, the set of states the current one represents must be a subset of the set of states of the basic one. This holds when

1) there exists a substitution mapping variables of the basic configuration to expressions, such that it transforms the threads and objects of the basic configuration to those of the current one, and
2) the set of restrictions of the basic configuration is a subset of the restrictions of the current one.

In the example, the first requirement is satisfied by a substitution $\{i_1 \mapsto i_1+1\}$; the second also holds. The substitution becomes an assignment statement of the residual program:

```
i1 = i1+1;
```

By adding the assignment and the goto M1 to the residual program, the supercompiler completes the positive branch of the loop condition.

Residual program:

```
        public static void doit(int k) {
//      Initial configuration.
            i1 = 1;
//      Basic configuration.
        M1: if (!(i1<=k1)) goto M2;
                int n1 = My.readInt();
                int m1 = My.F(n1);
                My.writeInt(m1);
                i1 = i1+1;
                goto M1;
//      Configuration like the basic one but with restriction {i1 > k1}.
        M2:
//      ...
        }
```

The supercompiler returns to the configuration on the negative branch (labeled M2), which was laid aside after generating the if statement, and transiently drives it to the end without adding any new statements to the residual program. Supercompilation ends. The final result is as follows.

Residual program:

```
        public static void doit(int k) {
//      Initial configuration.
            i1 = 1;
//      Basic configuration.
        M1:  if (!(i1<=k1)) goto M2;
                    int n1 = My.readInt();
                    int m1 = My.F(n1);
                    My.writeInt(m1);
                    i1 = i1+1;
                    goto M1;
        M2:
        }
```

Since Java has neither the notion of label, nor goto statement, the residual program must be postprocessed. All gotos should be either folded into available Java control constructs, or procedures with goto should be split into several copies, each goto becoming an auxiliary procedure call. The absence of goto is a serious drawback of a language intended for program transformations. Unfortunately, the authors of Java did not think about such use of Java.

Here is the ultimate residual program after postprocessing.

Residual program:

```
public static void doit(int k1) {
    for (int i1=1; i1<=k1; i1++) {
        int n1 = My.readInt();
        int m1 = My.F(n1);
        My.writeInt(m1);
    }
}
```

Conclusions

The example has demonstrated almost all techniques of supercompilation which we plan to implement in the first version of the Java supercompiler. Most are general methods; a few are specific for Java (loop syntax):

1. The notion of *configuration*:
    - A *configuration* is
        - a *set of states* of the original program, as well as
        - a *program point* of the residual program;
    - *Configuration variables* are
        - *parameters* of a configuration , as well as
        - *program variables* of a residual program;
    - The *configuration structure* resembles the representation of program state in the Java interpreter and consists of
        - a set of *threads*, and
        - a set of *objects;*
    - Additional elements in configurations but not interpretation structures are
        - *configuration variables* occurring instead of unknown values or the unknown recursive tail of an object,
        - *restrictions*, which are predicates on configuration variables,
        - the *safe/unsafe tag* for objects:
    - Each *initial configuration* supplied by a user generates a task for supercompilation.
2. The process of *driving:*
    - *transient driving:* threads are executed as if by an interpreter whenever configuration variables do not interfere. The results of some operations are evaluated even if they depend on configuration variables and are represented by symbolic expressions.
    - *residualization* of an unconditional statement: when a statement cannot be executed because it depends upon configuration variables or unknown procedures, a copy of it is put into the residual program. Program variables in the copy are replaced by their values, possibly involving configuration variables. If needed, an unknown result of evaluation of an expression is represented by a new configuration variable if it cannot be represented by a symbolic expression.
    - *branching* (*residualization* of a conditional statement): when the condition in an if statement cannot be evaluated to true or false, it is residualized and the process of supercompilation continues along one of the branches, usually the positive one first. When supercompilation of the first branch has completed, the second branch is processed. Other conditional statements, for, while, case, are compiled according their semantics by reduction to if.
    - *next configuration(s)* after residualization is (are) at worst the same as the one(s) before. However, any additional information revealed during a step is represented in the configuration, improving the residual program. Driving rules for each Java operator algorithmically define how a configuration is *narrowed* after a step by one of two means:
        - *contraction* of configuration variables: replacing these by constants or symbolic expressions involving other or new configuration variables;

- adding an elementary predicate to the set of *restrictions* of the current configuration. A class of elementary predicates is to be chosen by the author of a supercompiler. For Java, we plan to use inequalities, $\neq$, $<$, $\leq$.

3. *Configuration analysis:*
   - The goal of supercompilation is to construct a *finite* residual program. To achieve this, a supercompiler analyzes the trace of driving, compares the current configuration with previous ones, and takes the following decisions:
     - *looping back:* when the current configuration is a subset of an old one and can be reduced to it by a substitution, appropriate assignment statements reflecting the reduction and a goto[5] statement are put into the residual program. Additional rules may preclude looping back in order to construct a more efficient specialized program.
     - *generalization:* when the supercompiler finds an old configuration which has threads and objects with the same structure as the current configuration, except for variable values, it throws away the part of the residual program starting from the old configuration, builds the *least common generalization* of the configurations and makes it next after the old one. The corresponding assignments to the new variables of the generalized configuration are put on the arc from the old to the generalized configuration. Then the supercompiler reconstructs the residual program starting from the generalized configuration. This back-tracking is the main reason that supercompilation is often time-consuming.
     - *cutting*[6] the current or one of the previous configurations into two. This corresponds to a procedure call in the residual program. The two parts are supercompiled separately.
   - To determine particular configurations to loop back, to generalize or to cut is the hardest problem of supercompilation. As a first approximation, we will use the method described in [Turchin 88] as modified for multiple threads. Its idea is that only the structure of threads, not the values bound to program variables, is used to make decisions. This method guarantees termination of supercompilation.

4. *User control:*[6] Even if the supercompiler is very clever, it is just a machine, and the human user can substantionally help it to catch basic configurations, thus either improving the residual program, or drastically decreasing supercompilation time. However, this is not a task for the end-user, who knows nothing about supercompilation, and is just a programmer using Java source code libraries to write an application in Java. Authors of libraries of reusable software must think about program transformations. We plan to develop the means to annotate library programs with *pragma* information for the supercompiler. The supercompiler can be given some kind of description of preferable classes for basic configurations. This may include source program points, which may trigger the supercompiler to trace and compare configurations, or to perform some actions suggested by the user.

5. *Libraries of reusable software:* Supercompilation, as well as other program transformation techniques, allows new methods for structuring programs, in particular, by active use of interpreters. In order to bring supercompilation into practice, we plan to develop some demo libraries, including one to support the construction of data base applications

[5] Postprocessor will be used to fold gotos into the Java loop constructs or to translate to auxiliary function calls.
[6] This was not demonstrated by the example.

References

[GlÚck, Klimov 95]

R. GlÚck, A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree, *Proceedings of International Static Analysis Symposium*, Padova, Italy. LNCS, Vol. 724, pp. 112-123, Springer-Verlag, 1993.

[GlÚck, Klimov 95]

R. GlÚck, A.V. Klimov. Metasystem transition schemes in computer science and mathematics, *World Futures: the Journal of General Evolution*, Vol. 45, pp. 213–243, OPA, 1995.

[GlÚck, Klimov 97]

R. GlÚck, A.V. Klimov. On the degeneration of program generators by program composition, accepted to *New Generation Computing*, 1997.

[Jones et al 93]

N.D. Jones, C.K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice Hall: New York, London, Toronto 1993.

[Klimov 97]

A.V. Klimov. Specification of a class of supercompilers, submitted to *ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, June 12–13, 1997, Amsterdam, The Netherlands.

[Romanenko 90]

S.A. Romanenko. Arity raiser and its use in program specialization, in: Jones N. D. (ed.), *ESOP '90*. (Copenhagen, Denmark). LNCS, Vol. 432, pp. 341-360, Springer-Verlag, 1990.

[S›rensen et al 94]

M.H. S›rensen, R. GlÚck and N.D. Jones. Towards unifying deforestation, supercompilation, partial evaluation and generalized partial evaluation, in: D. Sannella, ed., *Programming Languages and Systems*, LNCS, Vol. 788, pp. 485–500, Springer-Verlag, 1994.

[Turchin 86]

V.F. Turchin. The concept of a supercompiler, *ACM Transactions on Programming Languages and Systems*, **8**, pp. 292–325, 1986.

[Turchin 88]

V.F. Turchin. The algorithm of generalization in the supercompiler, in: D. Bj›rner, A.P. Ershov, N.D. Jones, eds. *Partial Evaluation and Mixed Computation, Proceedings of the IFIP TC2 Workshop*, pp. 531–549, North-Holland Publishing Co., 1988.

[Turchin 93]

V. F. Turchin. Program transformation with metasystem transitions, *Journal of Functional Programming*, 3(3): 283–313, 1993.

[Turchin 96]

V.F. Turchin. Metacomputation: metasystem transition plus supercompilation, in: *Partial Evaluation. LNCS*, Vol. 1110, pp. 481–510, Springer-Verlag, 1996.

[Wadler, Odersky 97]

Ph. Wadler, M. Odersky. Pizza into Java: Translating theory into practice, in: *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, La Sorbonne, Paris, France, January 15-17, 1997

pJava
A Parallel Superset of Java for Automatic Parallelization

Andrei V. Klimov

Keldysh Institute for Applied Mathematics, Moscow

Larry D. Wittie

SUNY at Stony Brook, New York

Abstract

An extension of Java, called pJava, which allows for automatic parallelization and efficient program transformation is presented. The idea of pJava is to select a Java subset with suitable properties, and then gradually extend it while preserving the properties. The starting point of pJava is a purely functional subset of Java with data limited to immutable objects. Higher-level declarative notions will gradually be added to allow for easy, reliable, implicitly parallel programming. Ultimately mutable objects will be allowed as well, but under a special programming discipline. This discipline can be satisfied at a low-level by a qualified Java programmer, or at a high-level via constructs based on monotone objects, which can be easily applied by less-experienced users.

This research has been supported by the Office of Naval Research via grant 00014-96-1-0800.

Contents

Introduction

We propose to develop a language based on Java for easy parallel programming in networks. We call it *pJava*. The main reasons and goals for this work are as follows.

Why Java as the basis?

- Java is rapidly becoming a wide-spread language used by millions of programmers. A language that has as many features in common with Java as possible will be easier to learn and to use. For the same reason the Java authors based their work on C.
- Java already has good basic properties for automatic parallelization and program transformation. Its authors have done a good job of taking C and removing its poorly defined parts. In particular, the Java data model is good.

Why not Java? Why a new dialect?

- Java is a *low-level* language. It was not intended for automatic parallelization. It has an *explicit* method for parallel programming, that is, the notion of *threads*.
- The level of pJava will be *higher* than that of Java. Programs in pJava will be suitable for *automatic parallelization*, both *static* (at compile time) as well as *dynamic* (at run time). The method for parallel programming will be *implicit* and simpler to port to new computers.

What is the relation of pJava to Java?

- First, we select an almost universal *subset* of Java, which supports program transformation and is suitable for automatic parallelization. The subset is approximately a *single-assignment functional language*. It is *value-oriented* rather than *object-oriented*. Only *immutable*[1] objects are allowed to model complex values.
- Second, we *extend* it with *high-level* constructs, which do not interfere with the goal of automatic parallelization. Extensions are done in cycles, gradually introducing new notions:
  - *Value-oriented:*
    - Notions from *functional programming*, including *recursive data* definitions of the form $x = f(x)$, the semantics of which requires *lazy evaluation*.
    - Higher level declarative notions from specific applicative areas, especially based on experience from the *Norma* project [Zadykhailo et al 96]
  - *Object-oriented:*
    - Restricted Java classes, which allow automatic parallelization. These are referred to as *monotone objects* [Klimov 91].
- Other Java means (e.g. threads and arbitrary non-monotone classes) will *not* be prohibited. These are regarded as *low-level*, and not recommended for ordinary users, but allowed (and actually needed!) for experienced programmers. The pJava system will contain an analyzer that will annotate a program by telling what parts of a program belong to "true" higher-level pJava, and what are lower-level. This is not just paying tribute to Java. It is our strong opinion that a language containing only high-level notions would be impractical. Moreover,

[1] An object is *immutable* if it does not change after a constructor has initialized it. This is the defining property of a class that can be checked by the pJava system.

the idea of a monotone object substantially requires that infinitely many monotone objects be programmed by low-level means.

What is the background of our research into automatic parallelization of pJava?

- Traditional methods of automatic parallelization based of functional, single assignment, data flow languages.
- The experience in parallelization of higher-level declarative languages (like Norma [Zadykhailo et al 96]).
- Our experience in constructing monotone object classes, which allow for automatic parallelization and dynamic program transformation [Klimov 91].
- Achievements in deep program transformation disciplines such as *partial evaluation* [Jones et al 93] and *supercompilation* [Turchin 86], and our results in applying these to Java.

What are the characteristic features of the approach?

- *Implicit* parallelism of higher-level language notions for most users, and *explicit* notions for parallel programming only for experienced programmers, creating low-level parts of libraries of reusable software.
- Development of *high-level* languages, up to *declarative*, for particular application areas, as an extension of a common language, Java. High-level language notions are much better parallelized and transformed than low-level ones.
- *Easy* parallel programming by the masses as the main goal. This goal is the reason for our preference of high-level notions. If efficiency and simplicity of use conflict, we prefer to sacrifice some efficiency to retain ease of use, keeping in mind that computer speeds almost double each year.
- Our approach differs from that of languages such as HPF(ortran), HPJava [PCRC 96], which are based on extending a language by explicit low-level parallel methods, intended for experienced programmers.
- Use of modern *program transformation* techniques like supercompilation and partial evaluation to optimize programs deeply and to support automatic parallelization. A motto: "Language properties required for automatic parallelization and efficient program transformation are the same." Languages well-suited for one goal are well-suited for the other as well. Hence, our simultaneous development of a language for both goals.
- Optimization of parallel programs not only by parallelization, but by "*sequencialization*" as well, that is, by fusing threads and parallel processes to form smaller numbers of threads and processes with fewer delays for interactions. We consider pJava a high-level language with highly parallel interpretive semantics. To improve its programs, we either extract this implicit parallelism by converting the interpreter to a compiler and mapping it onto a modern computer architecture, or reduce inefficient concurrency by fusing threads.
- Construction of *demo libraries* of reusable software (for example, for data base applications), making use of our methods. To bring the new methods into practice, the authors will develop programs that demonstrate the value of the methods.

pJava Language Structure

pJava is a superset of Java consisting of two parts:

- A *high-level* part, which contains no explicit parallel notions and is intended for easy programming and automatic parallelization. It is based on a suitable Java subset and then is gradually extended preserving the required properties. Call this "pJava proper".
- A *low-level* part, which is a subset of Java, which contains explicit notions for parallel programming.

Together, the low-level and high-level parts cover all of Java. Actually, the low-level part may be defined as containing all Java notions except those belonging to the high-level part. Hence, the user need not limit himself to pJava proper. A preprocessor will annotate each program telling which parts belong to the high and low levels. Such a preprocessor will not only help users, but also is part of an automatic parallelization tool for high-level parts of programs.

The high-level parts of pJava consist of several layers of notions, each more complex than the previous one and relying on it. The layers of pJava are split into two parts: value-oriented and object-oriented.

Value-oriented levels

The key idea of the value-oriented part is that *mutable* objects are prohibited. The values of data entities cannot change. Although object-oriented Java notions such as classes and inheritance are present at these levels, they are used only to model *values*, that is, *immutable* data.

Definition. A class and its objects are *immutable* if only its constructor sets, or otherwise changes, the value of its local variables, and local objects are also *immutable*.

Statement. There exists a Java program analyzer that can check whether a class is mutable or not.

Note. The `final` type modifier of Java supports the notion of immutable objects, and allows the user to supply additional information for the analyzer.

1st level

First, we select a *subset* of Java, that approximately corresponds to a *functional language*, and allows for automatic parallelization.

- **Data:** Simple Java types (numeric, characters, Boolean) and immutable classes. No arrays. Only *tree* structures can be constructed by these means; not graphs.
- **Control:** Procedures programmed in *single-assignment* style. No threads. Exceptions are allowed under certain conditions.

2nd level

Second, we change and extend the semantics of the 1st level, preserving the syntax.

Statement. The 1st level allows for the following changes from *sequential* to *parallel* semantics, preserving the upwards equivalence of the result of computation:

- Each class is modified to be a subclass of a special class `IsReady` that controls an additional *tag* for each variable telling whether its value is *ready*, or not. After a variable has been created, the tag equals *not_ready*. After the variable takes its value for the first time, the tag changes to *ready*. After this, the value does not change.

- Class `IsReady` has procedures, `put` and `get`, which are `synchronized` (in Java terms): `get` waits for *ready*, `put` sends *ready* signals after it has assigned a value to a variable.
- Each *procedure* call is executed by starting a *new thread*.

Statement. The extension of the semantics is a proper *extension*:

- If a program returns a result by the sequential semantics, it returns the same result by the parallel semantics.
- If a program returns a result by the parallel semantics, it may execute infinitely or reach deadlock under sequential semantics.

Thus, we come to the first extension of Java, which we consider the 2nd level.

Exploiting this semantics extension, new programs may be written to define complex data by means of recursive equations of the form $x = f(x)$, expressed textually as an ordinary assignment. For example, consider an immutable class to represent integer lists, `IntList`:

```
class IntList {
    ...
};
class IntNil extends IntList {
    IntNil() {};
    };
class IntCons extends IntList {
    int head;
    IntList tail;
    IntCons(int head, IntList tail) {
        this.head = head;
        this.tail = tail;
    }
}
```

Then assignments of the following form make sense:

```
IntList x = F(x);
```

For example, an `F` which defines `x` to be a list of natural numbers from 1 to 100 can be coded as follows:

```
IntList F(IntList x) {
    return new IntCons(1, x.inc().take(99));
}
```

where the methods `inc` (increment all integers in a list by 1) and `take` (take first `length` elements of a list) are declared in the class `IntList`:

```
class IntList {
    IntList inc() {
        return
            this instanceof IntNil ?
                this
            :   new IntCons(((IntCons)this).head+1,
                            ((IntCons)this).tail.inc());
    }
    IntList take(int length) {
        return
            this instanceof IntNil || length==0 ?
                (IntList) new IntNil()
            :   new IntCons(((IntCons)this).head,
                            ((IntCons)this).tail.take(length-1));
    }
}
```

The procedures `inc` and `take` can execute in parallel because of the extension of the semantics. Old Java syntax is used. No new syntax notions are needed.

3rd level

Here lies a large world of declarative languages, e.g., functional, logical, and constraint languages. First of all, we will develop a Norma-like extension of Java.

The value-oriented part of pJava will not allow construction of arbitrary data structures. The most complex structures that can be represented using tail recursion on immutable objects are *trees* and their simpler degenerate forms, such as linear lists. This limitation gives us a rational reason to include mutable data in further extensions of pJava. Their use is not just a matter of improving efficiency, as in the usual arguments against the use of functional languages. Mutable data are necessary.

Object-oriented level

Fortunately, a large number of classes exist which can be programmed by low-level means using `synchronized` threads, but preserve all properties essential for program transformation when used at the value-oriented level. These classes (and their objects) are referred to as *monotone* [Klimov 91].

The notion of a monotone class

Definition. A set of classes are called *monotone* if any Java program that uses the classes and is written so as to satisfy the restrictions of the value-oriented level meets the following properties:

1. *determinacy:* evaluating procedures in arbitrary order according to parallel semantics gives the same result;
2. *recomputability:* if a procedure is evaluated with the same arguments for a second time, it returns an equivalent result, and produces no new side effects.

The first property means that equivalent parallel codes can exist. It makes automatic parallelization possible. The second property allows reliable computations: if a processor on which a process was scheduled fails, the process may be rescheduled and started from the beginning. Our research into supercompilation has shown that the second property is very important for effective program transformations as well.

An example of a monotone class

The following class defines objects which behave like variables with values that can be assigned only once. It is the simplest monotone example.

```
class IntVar {
    int value;
    boolean ready = false;
    synchronized void put(int n) throws ContradictoryAssignment {
        if (ready && value != n) throw ContradictoryAssignment;
        value = n;
        ready = true;
        notify();
    }
    synchronized int get(int n) {
        if (!ready) wait();
        return n;
    }
}
```

Monotone object-oriented programming

In a similar way, a library of basic monotone classes can be constructed. These use low-level mechanisms like `synchronized` and `boolean ready`. The must be written by a qualified programmer. However, when one defines higher-level objects by using only monotone objects as building blocks and no explicit thread-oriented notions are used, the new objects are *automatically* monotone. Having libraries of monotone functions will allow complex application programs, which are parallel by construction, to be written by ordinary Java programmers.

Monotone classes allow construction of arbitrary complex data, up to *graphs*. Our experience has shown that, although monotone objects do not allow arbitrary change of state, reactive and dialogue systems can be programmed using a small number of basic monotone objects that are programmed by low-level means.

Conclusions

We intend to use this sequence of extensions to develop pJava from Java. It will be a functional language extended to include at least write-once semantics for data variables, but will retain the familiar syntax of ordinary Java.

Programs based on the functional subset of pJava will allow program transformation and automatic parallelization techniques to be used to form faster running versions of the code for specific purposes. For example, a fast code for multiplying 100x100 matrices can be derived from a generic library routine for multiplying NxN matrices. The routine can easily be ported for efficient execution on any machine for which a pJava interpretation or compilation system exists.

Writing programs in pJava will also allow graphical composition of pJava library routines to form fast running parallel solutions for the specific application needs of masses of users. Whatever code is produced by composition of standard pJava routines can be optimized to run faster, at least by eliminating the parts of the combined routines that are not used in a particular composite calculation. This need will grow as more and more people want to use the Internet to find fast answers to their own questions.

References

[Zadykhailo et al 96]

A.N. Andrianov, A.B. Bugerja, K.N. Efimkin, I.B. Zadykhailo. The specification of the NORMA language. Draft Standard. Preprint of Keldysh Inst. of Appl. Math. , Russian Academy of Sci., 120(1995), pp.1-50.

[Jones et al 93]

N.D. Jones, C.K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice Hall: New York, London. Toronto 1993.

[Klimov 91]

A.V. Klimov. Dynamic specialization in extended functional language with monotone objects. *Proceedings of Symposium on Partial Evaluation and Semantics-Based Program Manipulation,* New Haven, Connecticut. 199-210, ACM Press 1991.

[PCRC 96]

HPCC and Java -- A Report by The Parallel Compiler Runtime Consortium (PCRC). Incomplete Draft, May 12 1996, http://www.npac.syr.edu/users/gcf/hpjava3.html.

[Turchin 86]

V.F. Turchin. The concept of a supercompiler, *ACM Transactions on Programming Languages and Systems*, **8**, pp. 292–325, 1986.

[Wadler, Odersky 97]

Ph. Wadler, M. Odersky. Pizza into Java: Translating theory into practice, in: *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, La Sorbonne, Paris, France, January 15-17. 1997.

The report on the subject

Creation of Efficient and Portable Parallel Programs
(grant ONR N00014-96-1-0800)

Contents

1 Technical background of the NORMA language creation.

1.1 The first works.

The first publications appeared in late 50s - early 60s [1,2]. The main idea of the language later named NORMA is very simple. It was an attempt to automate the design of the programs based on the jobs prepared by applied mathematicians from Keldysh Institute of Applied Mathematics for further programming. Usually those jobs were the result of applying numerical methods (more often grid method) to physical problems' solution. The intention was to create the language of jobs' specification corresponding to the constructions obtained after mathematical solution of the problem.

At first the authors called such method of specification "parameter record" later they used term "nonprocedural specification" nowadays term "declarative specification" is often in use.

Having analysed the material on this subject the authors came to the conclusion that analysis of index dependencies appeared in the relations between variables when continual equations are discretised would be the fact of great importance in designing program on declarative specification.

Let's consider briefly the first attempt to analyse such relations. There settled the task to analyse specifications of the following type:

$$\begin{aligned} \overset{1}{X}_i &= f^1(\overset{1}{X}_{i-\Delta_{1,1}}, \overset{2}{X}_{i-\Delta_{1,2}}, \ldots, \overset{n}{X}_{i-\Delta_{1,n}}) \\ \overset{2}{X}_i &= f^2(\overset{1}{X}_{i-\Delta_{2,1}}, \overset{2}{X}_{i-\Delta_{2,2}}, \ldots, \overset{n}{X}_{i-\Delta_{2,n}}) \\ &\ldots \\ \overset{n}{X}_i &= f^n(\overset{1}{X}_{i-\Delta_{n,1}}, \overset{2}{X}_{i-\Delta_{n,2}}, \ldots, \overset{n}{X}_{i-\Delta_{n,n}}) \end{aligned} \qquad (1)$$

It should be noted that the problem of parallelism wasn't urgent at that time because there were no proper computers that's why they solved the problem of automatic design of the loop process for computations set by relations (1). One can see that a relation is a construction where the variable with indices is placed in the left part and the function from the variables with indices in the right part. In this particular case the variables have only one index and displacement $\Delta_{k,l} \quad k,l = 1,\ldots,n$ is an integer constant. General form of relations will be considered below.

Let's consider that the values of variables $\overset{p}{X}_i \quad p = 1,\ldots,n$ are to be calculated for the value of one-dimensional domain set by integer numbers in the ranges from $m^0(p)$ to $m^N(p), \quad m^0(p) >> m^N(p), \quad p = 1,\ldots,n, \quad m^0(p)$ - boundaries of the calculated values. Let's also assume that all the values to the left of $m^0(p)$ are known. The last assumption is lies in the fact that a full-connected graph corresponds to the dependencies between the variables. The arcs of the graph are considered to connect the graph's node $\overset{k}{X}$ with node $\overset{l}{X}$ if $\overset{k}{X}$ is in the left part of the relation and $\overset{l}{X}$ is in the right one. We consider the full-connected graph because it has many difficulties in solving.

Matrix H with elements $h^t(p,q) = m^t(q) - m^t(p) + \Delta(p,q), \quad p,q = 1,\ldots,n$ is the main object the method of solution is based on. The variables with index t are the current state of the boundaries after step t. The values of the components which have the arguments (i.e. they have been already computed or set as the initial values) are assumed to be calculated at every step. Note that $h^t(p,q)$ is the number of components' values which could be computed for variable $\overset{p}{X}$ if it is depended only on $\overset{q}{X}$.

Let's give name "chain" to any sequence $u_1, u_2, \ldots, u_k$ from k numbers, $k \le n$ where all u_k are different. The sequence of the following type from the elements of matrix H may be built.

$$h^t(u_1,u_2); h^t(u_2,u_3); \ldots, h^t(u_{k-1},u_k); h^t(u_k,u_1).$$

Let's name sum of all the elements of chain z weight $p^t(z)$ of this chain and variable $S^t(z) = p^t(z)/k$ reduced weight. It is easy to see that p^t and S^t for every chain don't depend on the system of the boundaries and determined by the index displacements only:

$$S^t(z) = \frac{p^t(z)}{k} = \frac{\Delta_{u_1,u_2} + \Delta_{u_2,u_3} + \ldots + \Delta_{u_{k-1},u_k} + \Delta_{u_k,u_1}}{k}$$

Corresponding methods of research in organising the loop process of computations is developed in [2].

Let's generalise the results obtained in [2]. Let the chain with the minimal reduced weight S_M be found (an algorithm of determination of chain S_M is given in [2] and has estimate $cn^3 \log_2 n$, c - constant) then the following theorem is correct:

Theorem. *If the minimal reduced weight of the chain from (1) $S_M \le 0$ then the computation is impossible. If $S_M \ge 0$ then the computations may be organised in the way that $S_M \cdot n$ values of the variables on the average could be computed at every step.*

Here the term "on the average" is used in the sense of the possibility of computing different numbers of the variables at different steps in the case of non-integer S_M but np values of the variables are to be computed for k sequential steps.

Corollary 1. *Mere sequential computation appears in the case when S_M is reached on the chain with the length n and weight 1, i.e. in the system with the minimal reduced weight $S_M = 1/n$.*

Corollary 2. *Concurrent computation of every variable's values at step k is possible when $S_M \ge 1$.*

Corollary 3. *If $S_M \ge 1$, S_M is integer then S_M values of every variable can be computed at very step. The regular computation similar to this one may be organised neglecting the fractional part if S_M isn't integer.*

It should be noted that though the results given above follow simply from work [2] they are original and show the influence of index dependencies on the possibilities of parallelising. Authors consider the research done on this grant and obtained results very useful.

Given methods didn't have any influence on the design of the translator from the language of declarative specification as A.N. Andrianov constructs general method of computation process organization (parallel, sequential, parallel-sequential) for the systems of relations in rather general form (multidimensional case for the computational domains with complicated configuration is considered). Theoretical results and program implementation of some algorithms based on these results have been obtained.

One more note should be done. It is clear from the task's statement that we'll consider extracting natural parallelism represented in the relations. We aren't going to use any artificial methods dealt with e.g. transformation of the program. E.g. it is generally known that the elements of some tasks can be reduced to the form:

$$y_i = a_i * y_{i-1} + b_i,$$

where the values of a_i and b_i on grid [1,n] and initial value y_0 are assumed to be set. y_i is to be computed on grid [1,n]. The computation by our methods will be evidently sequential. Really here n=1 and $S_M = 1$ so the only one value will be computed at every step.

But it is well-known that there exists method of parallelising such a relation. Authors never refuse to include such methods and even parts of the programs in other languages into translating system. Further more the problem of including Norma extracts into other languages (we mean FORTRAN versions) should be discussed.

The authors also faced the problem of limiting the capabilities for parallelising (the problem of parallelism granularity) or imposing the constraints on the order of computations (e.g. caused by the requirements to the precision of computations).

To conclude this part we should note that its aim was to report some details about the history of the approach to the NORMA language design and to explain principal notions and problems appearing in the solution process on simple examples.

1.2 The advantages of the approach based on the NORMA language application.

1.2.1 Extremely high level of the language. New level of stability

We have gained much experience in designing complicated program systems and translators, now it is time to create really friendly programming languages. We shall take a decisive step and turn from universal languages to the languages for users to formulate problem solution in generic terms. We are sure that universal languages may be friendly only to system programmer. Hence we shall bend every effort to creation of specialized language for each application domain.

It is clear that such a language will be changed as soon as the application domain is changed. So extreme level of the language provides new level of the language stability. Let's note that the languages of universal type are constantly being specified to get a new level of conveniences in programming or to consider new computer resources. Constant modernisation of the FORTRAN language is a typical example. The idea of creating specialized languages has been known for a long time but it is time to pay much more attention to this approach. We consider this subject to become first and foremost in the programming.

We know by experience that terminology (languages) used by technical experts in the complex application domains gives us a good opportunity to consider the peculiarities of computational environment. The systems of notions in the application domain has been always based on the mathematical notions and it explains this phenomenon. It is worth to mention that there is no notion "memory" in such languages hence there is a principle of single assignment.

1.2.2 Portability

As the specification in the NORMA language contains full specification of an algorithm and doesn't reflect any peculiarities of the computer. It may be implemented on the computer with any architecture both sequential or parallel. Synthesising translator must be automatically adapted to the peculiarities of the architecture or must allow for the peculiar architecture.

1.2.3 New level of programming reliability

As the user is free from the necessity of making a program (this part is carried out by a translator) then such process of programming doesn't cause mistakes (up to the reliability of a translator). Further more besides the traditional syntactical and semantic diagnostics the synthesising translator can send messages about the errors in the essential notions. E.g., impossibility of organising computation caused by insufficient initial data, by the mistakes in the index displacements etc.

1.2.4 Efficiency of the programs

High efficiency of automatically designed program is based on the capability of deep parallelising and providing the necessary level of parallelism granularity. Generally speaking the synthesising translator has the capability of estimating the different variants of the representation of the declarative specification in the program and finding of the best one according to the built-in rule or in the dialogue with the system programmer or the user himself.

1.3 Particularising the statements on the example.

There some characteristics of the language only in connection with the example given below will be briefly described. More details about the language and the state of work are given in the next part.

The example given below is taken from the article [4] where the author, Lesly Lamport uses this example to explain the method of hyperplanes applied to the automated parallelising of loops:

```
      DO 99 I=1,L
      DO 99 J=2,M
      DO 99 K=2,N                                                    (2)
      U(J,K)=(U(J+1,K)+U(J,K+1)+U(J-1,K)+U(J,K-1))*0.25
   99 CONTINUE
```

This extract is a simplified variant of difference scheme for solving the task of Laplace on rectangular grid (J,K) by fixed number of iteration on I. The constraints imposed by this construction on the computations' order require the analysis of the computations' order (which is set by the nesting of loops), the analysis of variables' values(which is necessary because of the reassignment occurred in carrying out the program), and also the analysis of index dependencies. All these facts must be taken into consideration for the equal results both in parallel and sequential variants.

In the result of the analysis we can state that the values of the variables having index displacements J-1 and K-1 are taken from iteration I and the rest ones from iteration I-1 in the computation of the values in some point of 3-dimensional space with coordinates (I,J,K) (though only plane (J,K) is set explicitly, variable I introduces one more coordinate implicitly). All the values of **U** are considered to be known on all the boundaries and independent of I. Thus at the first step only the values of variable **U(2,2)** when I = 1 can be calculated ; at the second step - **U(2,3)** and **U(3,2)** when I = 1; at the third - **U(2,4)**, **U(4,2)**, **U(3,3)** when I = 1 and **U(2,2,)** when I=2. They proved that there can be found the family of the planes where the values of variable **U** in all the points belonged to the next plane in parallel and independent way can be calculated at every step (details see in the article by Lamport).

Note that this extract may be represented in the following form:

```
U(J,K,I)=(U(J+1,K,I-1)+U(J,K+1,I-1)
                      +U(J-1,K,I)+U(J,K-1,I))*0.25                   (3)
                      J=1,...,M;  K=2,...,N;  I=1,...,L.
```

Up to the notations this variant is a generic mathematical specification where iteration index is consider in a natural way. Besides the specification doesn't contain reassignments of the values to the variables (reuse of memory). So in extracting parallel actions on the specification (3) it isn't necessary to take into consideration neither this fact (we deal with the language with single assignment) nor the order of computing loop iteration. Only the problem of analysis of index dependencies is left.

As in such a description the mathematician sees the geometrical figure - parallelepiped then he mustn't forget the to set known values on the boundaries (the faces of parallelepiped) required for the computation or he will take care of their computation. Even if he has forgotten about them then the message about the error may be sent in the essential terms with the reference to the corresponding face.

Now let's consider the simplest variant of the task's solution scheme:

```
U(J,K,I)=(U(J+1,K,I-1)+U(J,K,I-1)
                      +U(J-1,K,I-1)+U(J,K-1,I-1))*0.25               (4)
                      J=2,...,M;  K=2,...,N;  I=1,...,L.
```

This variant is obtained in natural way when the values at a new iteration is obtained by the values at the previous iteration. This variant allows wide and evident parallelising (all the points at a new iteration level may be calculated simultaneously).

The second variant complicates finding the parallelism and decreases the available level of parallelising. This variant of specification may be explained either by the intention of algorithm optimization by means of saving the memory or by the wish of decreasing the number of iteration or by both. This approach is acceptable for sequential realization. But loosing the capability for parallelising is may be fatal in the parallel processing environment. Hence the idea of systematical transit from the sequential program to the parallel one (e.g. in HPF [5]) doesn't always produce the acceptable results. On the contrary we consider systematical transit from the specification allowing maximum parallelising to the variants optimising the necessary characteristics by decreasing the level of parallelising rather promising approach. Note that algorithm for parallelising for form (3) may give (in the case of memory optimization) the same level of parallelising and memory capacity as the method of hyperplanes by Lamport for extract (2).

Nowadays there are adherents of the systematical transit from the sequential programs to parallel ones. It refers to some extant to the supporters of HPF approach mentioned above. But there are some opponents of such a method. Let's cite the extract from the article by P.H. Welch, G.R.R. Justo [6]:

- design standards that exclude parallelism also exclude security for complex applications. This leads to growing losses - both financial and human life;
- efficient and robust systems cannot be built by "first getting them to work serially on one processor" and then "parallelising" them ;
- existing "dusty-deck" codes that represent massive financial investments that "cannot afford to be wasted", also represent massive serial codes that are becoming unmaintainable and are certainly unverifiable. These are technical dead-ends - as commercial pressures will gradually make clear to all those who persist with them;
- tools to assist the parallelization of large-scale serial code are very difficult to make, will be very expensive to buy and will not be needed by the time they are half-made to work.

From our point of view instead of transit from the sequential version of the program to the parallel one or direct design of the parallel program there is a more promising third way when the original statement of a problem may be realized both in parallel and sequential variants. This approach is based on the main principle: *when formulating a new task don't impose extra constraints - further you may face such an environment where they cause inconvenience and inefficiency.* Fortunately mathematical specifications are kept to this principle nearly always.

1.4 Principal notions and constructions of the NORMA language

The NORMA language was originally created for specifying the problems of mathematical physics by difference methods. A particular system of notations is formed in this application domain. It is used by technical expert in writing formulae obtained in the process of working out solution's method.

This system of notations contains both *common* mathematical notions (e.g. integer, real numbers, vectors, matrices, functions and rows) and the notions *typical* for the given application domain (e.g. grid, index space, grid function, iteration on time and space).

Some abilities of the NORMA language are demonstrated below. More detailed specification is given in [7].

The notion *domain* is introduced in the NORMA language for representation of index space. Domain is a complex of integer sets $\{i_1,\ldots,i_n\}, n > 0$ each of them sets coordinates of the point from n-dimensional index space. Unique name - *index name* (name of index space's coordinate axe) is linked to every direction (coordinate axe) of n-dimensional space. Particular case of the notion domain is *grid* - logical rectangular domain.

One-dimensional domain is used for setting the range of the points on some coordinate axe of index space. The *name of one-dimensional domain*, the *name of index* and the *range* of changing index values are indicated in the simple case of declaration of one-dimensional domain:

```
RegionK: (k=1..15).
```

The boundaries of the range may be set by *constant expressions*, e.g.

```
GridMN: (i=M+3..2*N).
```

Multidimensional domain is built by operation ";" of domains' product. The example of two-dimensional domain's declaration is given below. Two-dimensional domain is obtained by domains **AxisK** and **AxisL** product:

Square: (AxisK: (k=1..15) ; AxisL: (l=1..5)).

Modification of a domain includes adding some points, deleting some points or changing the range.

Domain may be *conditional* and *unconditional*. Conditional domain consists of the points from index space which number and coordinates may be changed depending on satisfying or failure of satisfying the *conditions on domain*. Unconditional domain consists of the points from index space which number and coordinates may be determined at the translation stage.

It should be noted that the domain determines the values of index space points' coordinates but not the values of calculated variables in these points.

Scalar variables (scalars) are used in NORMA for representation of simple variables but *variables defined on domain* are used for representation of vectors, arrays and matrixes. Declaration of the variable sets its type - **REAL**, **INTEGER**, or **DOUBLE** and the variable on domain is connected with the domain indicated in the declaration (i.e. the values of this variable may be computed in every point of this domain). Declarations

VARIABLE First, Last DEFINED ON Square

defines variables **First**, **Last** on domain **Square**; it means that the values may be assigned to these variables in every point of domain **Square** for $k = 1,...15$, $l = 1,...5$.

Calculating formulae obtained by technical expert are usually written in the form of *relations*. E.g. calculating formulae for the solution system of linear equations by method of Gauss-Jordan has the form:

$$m_{o,i,j} = a_{i,j}, \qquad j = 1,...,N, \qquad i = 1,...,N;$$

$$r_{o,i} = b_i, \qquad i = 1,...,N;$$

$$m_{t,i,j} = m_{t-1,t,j} / m_{t-1,t,t}, \qquad j = 1,...,N, \qquad t = 1,...,N;$$

$$r_{t,t} = r_{t-1,t} / m_{t-1,t,t}, \qquad t = 1,...,N;$$

$$m_{t,i,j} = m_{t-1,i,j} - m_{t-1,i,t} * m_{t,t,j},$$
$$j = 1,...,N, \qquad i = 1,...,t\text{-}1,t\text{+}1,...,N; \qquad t = 1,...,N;$$

$$r_{t,i} = r_{t-1,i} - m_{t-1,i,t} * r_{t,t},$$
$$i = 1,...,t\text{-}1,t\text{+}1,...,N; \qquad t = 1,...,N;$$

$$x_i = r_{N,i}, \qquad i = 1,...,N;$$

Extract from the NORMA program is given below:

```
Ot:(t=0..n) . Oi:(i=1..n) . Oj:(j=1..n).
Oij:(Oi;Oj) . Otij:(Ot;Oij) .
Oti:(Ot;Oi) . Otij1:Otij/t=1..n.  Oti1:Oti/t=1..n.
      DOMAIN PARAMETERS n=10.
VARIABLE a DEFINED ON Oij.  VARIABLE m DEFINED ON Otij.
VARIABLE b DEFINED ON Oi.  VARIABLE r DEFINED ON Oti.
      INPUT a ON Oij,  b ON Oi.
      OUTPUT x ON Oi.
FOR Otij/t=0 ASSUME m = a.
```

FOR Oti/t=0 ASSUME r=b.
OtiEQtj1, OtiNEtj1:Otij1/i=t. OiEQti1, OiNEti1:Oti1/i=t.
FOR OtiEQtij1 ASSUME m = m[t-1,i=t]/m[t-1,i=t,j=t].
FOR OiEQti1 ASSUME r = r[t-1,i=t]/m[t-1,i=t,j=t].
FOR OtiNEtj1 ASSUME m = m[t-1]-m[t-1,j=t]*m[i=t].
FOR OiNEti1 ASSUME r = r[t-1]-m[t-1,j=t]*r[i=t].
FOR Oi ASSUME x = r[t=n].

Necessary computations are described by **ASSUME** operator

FOR *domain* **ASSUME** *relation*.

This operator is a key-notion of the NORMA language. The relation set the rule of computing the variable's value from the left part on the values of the variable from the right part and index dependencies between the variables. The variable from the left part is to be computed in all the points of the *domain*; the rule for this computation is defined definitely but immediate fulfilment of the computation isn't required in the given place of the program and the method and the order of the computation isn't set. Some formal similarity with assignment operator mustn't mislead you.

Indices without displacements in the formulae notations may be omitted because they are automatically restored by the translator in analysing the program.

There are also conditional domains used in the considered extract. Thus definition **OtiEQtij, OtiNEtij1:Otij1/i=t** determines two disjoint subdomains **OtiEQtij1** and **OtiNEtij1**. The first subdomain consists of the points from modified domain **Otij1** where given condition **i=t** is true and the second of the points where this condition is false (i.e. **i≠t**). In general case the condition in declaration of conditional domains is represented by logical expression.

The differences in the given above two ways of writing calculating formulae are in the form of writing (index representation, linearity of the specification) but they are equivalent in their contents.

Special instruction **ITERATION** is used in the NORMA language for the description of iterative processes.

Declarations of input or output variables are used in their usual way. E.g. declarations

B1,B2: B/z<Eps.
INPUT Velocity ON A. OUTPUT Tau ON B1.
INPUT X, ALFA.

are the requests for input of the values of scalars **X, ALFA**, variable **Velocity** in all the points of domain **A** and also for output of the value of variable **Tau** in all the points of domain **B** where condition **Z<Eps** is satisfied.

The order of input or output of the variables' values isn't defined, it is determined in the process of translation of the program automatically (it requires special processing of the data files by the translator).

To finish the introduction of some NORMA facilities we consider reduction functions. They are analogies of the traditional mathematical notations of $\sum$, $\prod$ max, min type. Call to these functions has the form:

name-of-function ((name-of-domain) arithm-expression).

Domain determines the set of the domain's points where the function is to be computed, arithmetical expression is a set of the values which the function is to be applied to. Let's compute

$$V_i = W_i + \sum_{j=1}^{m} A_{i,j} * X_j \qquad i=1,...,n.$$

The description of this computation in the NORMA language

```
VARIABLE A DEFINED ON Grid : (Oi:(I=1..N);Oj:(J=1..M)).
VARIABLE V,W DEFINED ON Oi. VARIABLE X DEFINED ON Oj.
FOR Oi ASSUME V  =  W+sum((Oj)  A*X).
```

coincides (up to the notations) with the original calculating formula. Reduction functions' realization for different computational environments isn't an easy problem but this problem is solved by the translator not the user.

2 Semantics of the NORMA language

Formal semantics' specification of the NORMA language versions 1-22 [7] are given in this part. Semantics' specification is based on the application of operational approach [8,9] and formal methods of relational algebra [10,11].

2.1. Background information

2.1.1 Notations

There are some notations used below:

R - set of real numbers;
N - set of natural numbers;
I - set of integer numbers;
T - set of program P identificators;
F - set of program P arithmetical expressions;
L - set of program P logical expressions ;
$\varnothing$ - empty set;
Λ - value isn't defined;
$\bullet$ - operation of strings' concatenation;
$S_1 \propto S_2$ - relation of occurrence of substring S_1 into string S_2;
$\|A\|$ - power of set (or relation) A;
~ - "is by definition";
:= - "make equal";
$\mathbb{T}$, $\mathbb{F}$ - logical values **TRUE, FALSE**.
The construction of the choice has the form:

$$(P_1 \to O_1; P_2 \to O_2; \ldots P_n \to O_n)$$

where $P_1,\ldots,P_n$ - finite number of statements, $O_1,\ldots,O_n$ - any objects. It determines the first (counting from the left to the right) object O_i which has $P_i = \mathbf{T}$. Besides

- if all $P_i = \mathbf{F}$, $i = 1,..,n$ then the construction of choice is an empty object;
- if the variables included into P_i are used in setting the object O_i then the values assigned to these variables have $P_i = \mathbf{T}$;
- if terms of $\forall x F(x)$ type are included into P_i the usage of the variable x in setting object O_i means that all the values may be assigned to this variable (in the arbitrary order) when $F(x) = \mathbb{T}$;
- if terms of type $\exists x F(x)$ are included into P_i the usage of the variable x in setting object O_i means that *any* value may be assigned to this variable (but only one) when $F(x) = \mathbb{T}$.

2.1.2 The elements of relational algebra

Give some definitions according to e.g.[10].

$D_1,\ldots,D_k$ - some (may be identical) sets.

R is said to be a *relation* on sets $D_1,\ldots,D_k$, $k \geq 1$, if $R \subseteq D_1 \otimes \ldots \otimes D_k$ where $\otimes$ - operation of Cartesian product.

Sets $D_1,\ldots,D_k$ are called *domains* of relation R.

Elements $\langle d_1,\ldots,d_k \rangle$ of relation *R* which have $d_i \in D_i$, $i = 1,\ldots,k$ are called *tuples* of relation *R*.

Specification of type $R(A_1,...,A_k)$ are called *scheme* of relation *R* and A_i is an *attribute* of relation *R*.

Two types of operations are defined on relations: traditional operations on sets (union, intersection, Cartesian product etc.) and special relational operations (choice, filtration, projection, interconnection).

Let's introduce the following notations:

R_1, R_2, R - names of relations,

A - attribute of relation R,

C - logical condition,

α, β - tuples of relations.

$R_1 \otimes R_2$ - *operator of relations' Cartesian product.*

Arguments: relations R_1 and R_2.

Result: extended Cartesian product of relations R_1 and R_2. If $R_1 = \alpha_1,...,\alpha_n$, $R_2 = \beta_1,...,\beta_m$ then $R_1 \otimes R_2 = \alpha_1 \circ \beta_1, \alpha_1 \circ \beta_2,..., \alpha_1 \circ \beta_m,..., \alpha_n \circ \beta_1, \alpha_n \circ \beta_2,..., \alpha_n \circ \beta_m$.

R_1 **ADD-TO** R_2 - *operator of adding relation* R_1 *to* R_2.

Arguments: relations R_1 and R_2 compatible by union i.e. having equal numbers of attributes but for i=1,...,n the value of one and the same domain D_i is assigned to *i*-attribute of R_1 and R_2.

Result: relation R_2 added with the tuples of R_1 (without repetitions).

$R_1 - R_2$ - *operator of* R_1 *and* R_2 *relations' difference.*

Arguments: relations R_1 and R_2 compatible by union.

Result: relation R containing the tuples of relation R_1 which aren't included into relation R_2.

F(R,C) - *operator of filtration.*

Arguments: relation R and logical condition C built from constants and attributes of relation R by means of comparison operations $=, \neq, >, <, \geq, \leq$ and logical operations $\vee, \wedge, \neg$..

Result: relation consists of those tuples of relation R where condition C is true.

$R(A)$ - *operator of interception.*

Arguments: relation R and its attribute A.

Result: relation consists of the values of attribute A only in relation R (without repetitions).

$R(\overline{A})$ - *operator of projection.*

Arguments: relation R and its attribute A.

Result: relation, consists from the tuples of relation R where the values of attribute A (without repetitions) are deleted from.

$\alpha \circ \beta$ - *operator of tuples' interconnection.*

Arguments: $\alpha = \langle a_1,...,a_n \rangle$ and $\beta = \langle b_1,...,b \rangle_n$.

Result: tuple $\alpha \circ \beta = a_1,...,a_n, b_1,...,b_m$.

$R_1 \circ R_2$ - *operator of relations' interconnection.*

Arguments: $R_1 = \{\alpha_1, \alpha_2,..., \alpha_n\}$ and $R_2 = \{\beta_1, \beta_2,..., \beta_n\}$, R_1 and R_2 - relations with equal power

Result: relation $R = \{\alpha_1\beta_1, \alpha_2\beta_2,..., \alpha_n\beta_n\}$.

f(R) - *operator of relations' transformation.*

Argument: $R = \{\alpha_1, \alpha_2,..., \alpha_n\}$

Result: relation $R = \{f(\alpha_1), f(\alpha_2),..., f(\alpha_n)\}$.

UPDATE(α, β, R) - *operator of tuple's renewal.*

Arguments: relation R, tuple $\alpha = \langle \alpha_1,..., \alpha_n \rangle \in R$, tuple $\beta = \langle \beta_1,..., \beta_n \rangle$ where the values of the same domain D_i are assigned to α_i and β_i.

Result: relation R where tuple α is substituted by tuple β.

Let's consider arranged relation $\Pi(I)$, $I \in \mathbf{N}$ defined.

2.1.3. Principal notions and definitions.

References to syntactical notations introduced in specification of the NORMA language [7] by extended notation of Backus-Naur is used here. Further the grammar of the NORMA language will be denoted as G_{Norma} and the language engendered by this grammar $L(G_{Norma})$. Syntactical notations will be typed in italic e.g.: *main-part*, *declaration-of-domain*.

The process of $P \in L(G_{Norma})$ program computation is come to:

- transformation of P to standard form P_s by sequence $U_1,...,U_n$ of preprocessor's operators;
- design *abstract program* $A(P)$ on P_s, the abstract program is represented in the form of relations *Domains, Variables, Input, Output, InputData, Relations*;
- interpretation of abstract program $A(P)$ by abstract AM machine it is come to carrying out sequence $Q_1,...,Q_k$ operators of the relational algebra and modifying the relations given above. The process of interpretation consists of the sequence of *stages*. The definition of interpretation's stage is given below in the description of the rules of AM machine work.

Thus the semantic $Sem(P)$ of the program P is determined by the following process:

$$Sem(P): \quad P_s := Pre(P), \quad A(P) := Tr(P_s), \quad AM(A(P)),$$

where Pre is a preprocessor transforming P to equivalent standard form P_s;

Tr is a translator preparing abstract program $A(P)$ for AM machine on standard representation P_s. Functions of preprocessor Pre and translator Tr are defined in the next parts.

Operators U_j, $j = 1,...,n$ and Q_i, $i = 1,...,k$ may be *conditional* and *unconditional*. The transformation of program P is the result of carrying out unconditional operator U_j. Changing of relations' tuples is the result of carrying out conditional operator Q_j. Value $\mathbf{T}$ or $\mathbf{F}$ is the result of carrying out U_j or Q_j.

Program P contains *semantic error* if

$$\exists j: U_j = \mathbf{F} \vee \exists i: Q_i = \mathbf{F}$$

Program P is *semantically correct* if there is no semantic errors in it.

Variable X_1 of program P has *information dependence* on variable X_2 of program P if the value of X_2 or the value of X_3 which has information dependence on X_2 is necessary for the computation of X_1 value. Variable X_1 has *information independence* if there is no variable X_2 of program P which variable X_1 depended on. The value of X_1 is *directly computable* if it has information independence or the values of all the variables $Y_1,...,Y_n$ which X_1 is depended on have been already computed.

Abstract AM machine is a non-determined device: semantics $Sem(P)$ makes correspondence of semantically correct program P to the class of sequences made up by the operators of the relational algebra which performance by AM machine reflects all the possibilities of computations parallelization and gives identical results.

The parallelism determined only by information dependencies between the variables of program P is said to be *natural (ideal) parallelism* of program P. Informally natural parallelism means that at every moment the values of all the directly computable variables of program P are being computed. From this point of view semantics $Sem(P)$ may be considered as *the semantics of natural parallelism* as in particular AM realises the natural parallelism of program P.

Let's define the way of representing syntactically correct program P (syntactically incorrect programs aren't considered as they don't belong to $L(G_{Norma})$).

Define set **E** of *elementary objects*, set **H** of *compound objects* and set **O** of language $L(G_{Norma})$ *objects*:

E = {*identifier, key-word, constant, sign-of-operation, delimiter*};

H = {*program, part, declaration, operator, ..., declaration-of-domain, declaration-of-input,...,arithm-expression,...*}

$\mathbf{O} = \mathbf{E} \cup \mathbf{H}$

Informally set **H** consists of the language constructions engendered by all the non-terminals of grammar G_{Norma} except non-terminals included into **E** and *principal-symbols*.

Consider system (**O**,**S**), **S** is a set of *selectors*.

Let $is\text{-}\alpha$ be a predicate, defined on **O**, $is\text{-}\alpha : \mathbf{O} \to \{\mathbf{T},\mathbf{F}\}$. Selector of $\overline{is}\text{-}\alpha$ is said to be subdomain **O** satisfied predicate $is\text{-}\alpha$.

E.g.

$$is\text{-}identifier = \{x \mid is\text{-}identifier(x)\}$$

For the case when selector $\overline{is}\text{-}a$ contains the only element a let's use notation **a** (where it doesn't cause ambiguity). E.g. terminals $\overline{is}$-(and $\overline{is}$-**BEGIN** will be represented as **(** and **BEGIN** correspondingly.

Program P is corresponded to the system $(\mathbf{O}_P,\mathbf{S}_P)$, $\mathbf{O}_P \subset \mathbf{O}$, $\mathbf{S}_P \subseteq \mathbf{S}$ and it is represented in the form

$$P = o_1 \bullet o_2 \bullet ... \bullet o_n = o_1 o_2 ... o_n, \quad n \geq 1, \quad o_i \in \mathbf{O}_P \qquad (5)$$

and the application of selector $s \in \mathbf{S}_P$ to a program (or its part) in the form (2) is represented in the form $s(P)$ (correspondingly $s(o)$, $o \in \mathbf{O}_P$).

Let's introduce some auxiliary operators and functions.

$position(o_i,o)$ - function which determines the number of subobject o_i in object o for representation of (5) type

$$position(o_i,o) = i$$

Object o_i is said to be to the left of object o_j in object o (notation $o_i \prec_o o_j$) if $position(o_i,o) < position(o_j,o)$.

$Replace(o_1,o_2,o)$ - operator of substitution o_2 instead of o_1 in o ; $o_1,o_2,o \in \mathbf{O}_P$.

$$Replace(o_1,o_2,o) \sim (o = A \bullet o_1 \bullet B \to A \bullet o_2 \bullet B)$$

$ReplaceAll(o_1,o_2,o)$ - operator of substitution o_2 instead of all the occurrences of o_1 into o; $o_1,o_2,o \in \mathbf{O}_P$.

$$ReplaceAll(o_1,o_2,o) \sim (\forall o_1(o_1 \propto o) \to Replace(o_1,o_2,o))$$

$Delete(o_1,o)$ - operator of deleting the occurrence of o_1 into o ; $o_1,o \in \mathbf{O}_P$.

$$Delete(o_1,o) \sim (o = A \bullet o_1 \bullet B \to A \bullet B)$$

$DeleteAll(o_1,o)$ - operator of deleting all the occurrences of o_1 into o ; $o_1,o \in \mathbf{O}_P$.

$$DeleteAll(o_1,o) \sim (\forall o_1(o_1 \propto o) \to Delete(o_1,o))$$

$Include(o_1,o)$ - operator of including o_1 into o, keeping syntactical correctness of object o; $o_1,o \in \mathbf{O}_P$.

$uname(P)$ - generation of the name unique in program P.

$uname(P) = o, \quad o \in \overline{is}\text{-}identifier(\mathbf{O}) \wedge \neg\exists p(p \in \mathbf{O}_P \wedge o = p)$.

2.2 Relations of abstract program $A(P)$.

Domains (*Cond, D, Iname, I*): domains of program $A(P)$.

$Cond \in \mathbf{L}$ - logical expression.

$D \in \mathbf{T}$ - identifier of domain.

$Iname = I_1 \circ ... \circ I_n$, $I_i \in \mathbf{T}$ - identifier of domain's indices, n - dimension of domain D. $I \in D_1 \otimes ... \otimes D_n$, $D_i \in \mathbf{N}$ - index of domain D. The values of D domain (sets of integer numbers) are values I.

Tuple $\langle Cond \circ D \circ Iname \circ I \rangle = \langle Cond \circ D \circ I_1 \circ ... \circ I_n \circ i_1 \circ ... \circ i_n \rangle$ belongs to relation *Domains*, if point $i_1,...,i_n$ of index space $I_1 \circ ... \circ I_n$ belongs to domain D.

Variables (*D, X, Iter*): definition domains of program $A(P)$ variables.

$D \in \mathbf{T} \cup \varnothing$ - identifier of domain.

$X \in \mathbf{T}$ - identifier of variable.

$Iter \in \mathbf{T}$ - identifier of iteration index.

Tuple $\langle D \circ X \circ \Lambda \rangle$ belongs to relation *Variables* if variable X is defined on domain D. If variable X is indicated in the list of iterated variables with *iteration index Iter* then tuple of relation *Variables* has the form $\langle D \circ X \circ Iter \rangle$.

Input (*Var, Value, File*): input variables of program $A(P)$.

$Var = X(I)$, $X \in \mathbf{T}$ - identifier of an input variable, $I \in D_1 \otimes ... \otimes D_n$, $D_i \in \mathbf{N}$ - index of input domain D (set of integer numbers).

$Value \in \mathbf{R} \cup \Lambda$ - values of input variable X with indices I.

File - name of an input file.

Output (*Cond, Var, Value, File*): output variables of program $A(P)$.

$Cond \in \mathbf{L}$ - logical expression.

$Var = X(I)$, $X \in \mathbf{T}$ - identifier of an output variable, $I \in D_1 \otimes ... \otimes D_n$, $D_i \in \mathbf{N}$ - index of output domain D (set of integer numbers).

$Value \in \mathbf{R} \cup \Lambda$ - values of output variable X with indices I.

File - name of an output file.

InputData (*Var, Value, File*): the values of input variables of program $A(P)$, stored in the external files.

Attributes are coincided with the attributes of relation *Input* except the set of values of attribute *Value*: $Value \in \mathbf{R}$.

Relations (*Name, ItVal, Def, Cond, Var, Value, Func*): computed variables of program $A(P)$.

$Name \in \mathbf{T}$ - identifier of $A(P)$ program's part.

$ItVal \in \{ItId \bullet C\} \cup \{\varnothing \ \varnothing\}$ - character of computation in construction *iteration* ($ItVal = \{ItId \bullet C\}$) or outside *iteration* $ItVal = \{\varnothing \ \varnothing\}$; $ItId \in \mathbf{T}$ - identifier of *iteration index*. $C \in \mathbf{I}$ - the value of *iteration index*.

$Def \in \mathbf{L}$ - logical expression.

$Cond \in \mathbf{L}$ - logical expression.

$Var = X_1(I_1)...X_k(I_k)$, $k \geq 1$ - variables computed in part *Name*. $X_i \in \mathbf{T}$ $i = 1,...,k$ - identificators of variables; $I_i \in D_1 \otimes ... \otimes D_n$, $i = 1,...,k$, $D_i \in \mathbf{N}$ - the values of computation's domain $D_1 \otimes ... \otimes D_n$ (sets of integer numbers).

$Value = Value_1,...,Value_k$; $Value_i \in \mathbf{R} \cup \Lambda$ $i = 1,...,k$ - values of computed variables

$Var = X_1(I_1)...X_k(I_k)$, $k \geq 1$.

$Func \in \mathbf{F}$ - type of functional dependence for variables Var.

The tuples in every relation defined above are considered arranged in some fixed way.

2.3. Environment of abstract AM machine.

Environment of abstract AM machine

$Env = \{Domains, Variables, Input, Output, Relations, InputData, IndSpace, Files\}$

$IndSpace = \{IndSpace_i\}$, $i = 1,...,k$, k – number of program P parts;

$IndSpace_i = \underline{name\text{-}part}_i \bullet o_1^i \bullet o_2^i \bullet ... \bullet o_n^i$, $n \geq 0$,

$o_j^i \in \overline{is}\text{-}\underline{name\text{-}index}(part_i)$,

$Files$ - the names of external data files.

$Files = (filename_1,..., filename_l\}$, $l \geq 0$.

2.4 Definition of preprocessor $Pre(P)$.

Preprocessor Pre translates the text of some program P into equivalent standard representation P_s or fixes semantic errors.

$$
\begin{aligned}
Pre(P) \sim\ & \textbf{MAIN-EXIST}(P); \\
& \big(\forall o(o \in \overline{is}\text{-}part(P)) \rightarrow \\
& \qquad \textbf{LOCAL-VARIABLES-RENAME}(o); \\
& \qquad \textbf{NONREC-DOMAIN-DECLARATION}(o); \\
& \qquad \textbf{EXIST-UNIQ-DECLARATION}(o); \\
& \qquad \textbf{MACRO-INDEX-EXCLUDE}(o); \\
& \qquad \textbf{DEF-INDEX-SPACE}(o); \\
& \qquad \textbf{PAR-DOMAIN-EXCLUDE}(o); \\
& \qquad \textbf{CONST-EXPR-EVALUATE}(o); \\
& \qquad \textbf{STANDARD-DOMAIN}(o); \\
& \qquad \textbf{STANDARD-ASSUME}(o)\big)
\end{aligned}
$$

2.5 Specification of preprocessor's operators.

◊ *Checking main part existence* **MAIN-EXIST**(P).

Main part must be in the program and be the only one.

$$\textbf{MAIN-EXIST}(P) \sim \exists! o(o \in \overline{is}\text{-}main\text{-}part(P))$$

□ *Renaming local variables* **LOCAL-VARIABLES-RENAME**(o).

Names localized in the parts are substituted to the unique ones. The names of parts and functions aren't renamed. Localized in the parts names of input variables (scalars and variables defined on domain) are renamed according to the names of input variables set in the input files.

$$
\begin{aligned}
& \textbf{LOCAL-VARIABLES-RENAME}(o) \sim \\
& \quad \big(\forall p(p \in \overline{is}\text{-}\underline{name\text{-}main\text{-}part}(o)) \rightarrow; \\
& \quad \forall p(p \in \overline{is}\text{-}\underline{name\text{-}simple\text{-}part}(o)) \rightarrow; \\
& \quad \forall p(p \in \overline{is}\text{-}\underline{name\text{-}function}(o)) \rightarrow;
\end{aligned}
$$

$$\forall p(p \in \overline{is}\text{-}\underline{name\text{-}external\text{-}function}(o)) \rightarrow;$$
$$\forall p(p \in \overline{is}\text{-}\underline{name\text{-}external\text{-}simple\text{-}part}(o)) \rightarrow;$$
$$\forall p(p \in \overline{is}\text{-}\underline{name\text{-}reduction\text{-}function}(o)) \rightarrow;$$
$$\forall p(p \in \overline{is}\text{-}\underline{name\text{-}standard\text{-}function}(o)) \rightarrow;$$
$$\forall p(p \in \overline{is}\text{-}input\text{-}scalar(o)$$
$$\wedge\exists u(u \in \overline{is}\text{-}\underline{name\text{-}scalar}(p)) \wedge \exists v(v \in \overline{is}\text{-}\underline{name\text{-}file}(p)))$$
$$\rightarrow \textbf{LOCAL-IN-NAMES}(u, v, o, Files);$$
$$\forall p(p \in \overline{is}\text{-}input\text{-}scalar(o)$$
$$\wedge\exists u(u \in \overline{is}\text{-}\underline{name\text{-}scalar}(p)))$$
$$\rightarrow \textbf{LOCAL-IN-NAMES}(u, \text{'norma.dat'}, o, Files);$$
$$\forall p(p \in \overline{is}\text{-}input\text{-}on\text{-}domain(o)$$
$$\wedge\forall u(u \in \overline{is}\text{-}input\text{-}on\text{-}domain(p)) \wedge \exists v(v \in \overline{is}\text{-}\underline{name\text{-}file}(p)))$$
$$\rightarrow \textbf{LOCAL-IN-NAMES}(u, v, o, Files);$$
$$\forall p(p \in \overline{is}\text{-}input\text{-}on\text{-}domain(o)$$
$$\wedge\forall u(u \in \overline{is}\text{-}input\text{-}on\text{-}domain(p)))$$
$$\rightarrow \textbf{LOCAL-IN-NAMES}(u, \text{'norma.dat'}, o, Files);$$
$$\forall p(p \in \overline{is}\text{-}identifier(o)) \rightarrow ReplaceAll(p, uname(P), o)\Big)$$

Auxiliary function **LOCAL-IN-NAMES**(u, v, o, *Files*) carries out agreed renaming of the names of input variables u localized in part o and input from file v, and the names of input variables set in input files *Files*.

LOCAL-IN-NAMES(u, v, o, *Files*) ~
$$\Big(\exists p(p \in Files \wedge v = p \wedge \exists q(q \in \overline{is}\text{-}\underline{name\text{-}scalar}(p) \wedge q = u))$$
$$\rightarrow t := \textbf{UNAME}(P); ReplaceAll(q, t, p); ReplaceAll(u, t, o);$$
$$\exists p(p \in Files \wedge v = p \wedge \exists q \exists r(q \in \overline{is}\text{-}\underline{name\text{-}variable\text{-}on\text{-}domain}(p) \wedge q = u$$
$$\wedge r \in \overline{is}\text{-list-}range\text{-}of\text{-}index \wedge q \bullet (\bullet r \bullet)))$$
$$\rightarrow \textbf{RENAME-IND}(r, p, o)\Big)$$

Auxiliary function **RENAME-IND**(r, p, o) carries out agreed renaming of input variable's indices r localized in the part o and input from file p.

RENAME-IND(r, p, o) ~
$$(\forall w(w \in \overline{is}\text{-}\underline{name\text{-}index}(r)$$
$$\rightarrow t := \textbf{UNAME}(P); ReplaceAll(w, t, p); ReplaceAll(w, t, o))$$

◊ *Checking non-recurrence of domains' declarations* **NONREC-DOMAIN-DECLARATION**(o).

Domain with name o_1 is said to be depended on domain with name o_2 in part o if either the name of domain o_2 or the name of domain o_3 depended on o_2 is included textually into *declaration-of-domain* o_1. Let this character be checked by function **DEPENDENCE-DOMAIN** (o_1, o_2, o_3).

$$\textbf{DEPENDENCE-DOMAIN}(o_1,o_2,o_3) \sim$$
$$\exists p \exists q\Big(\quad p \in \overline{is}\text{-}declaration\text{-}of\text{-}domain(o)$$
$$\wedge\Big(\quad q \in \overline{is}\text{-}domain\text{-}product(p) \wedge p = o_1 \bullet\!:\!\bullet (\bullet q \bullet)$$
$$\wedge(o_2 \propto q \vee (o_3 \propto q \wedge \textbf{DEPENDENCE-DOMAIN}(o_3,o_2,o)))$$
$$\vee q \in \overline{is}\text{-}new\text{-}domain(p) \wedge p = o_1 \bullet\!:\!\bullet q$$
$$\wedge(o_2 \propto q \vee (o_3 \propto q \wedge \textbf{DEPENDENCE-DOMAIN}(o_3,o_2,o)))$$
$$\vee o_1 \in \overline{is}\text{-}name\text{-}of\text{-}conditional\text{-}domain(p)$$
$$\wedge(o_2 \in \overline{is}\text{-}name\text{-}of\text{-}domain(p)$$
$$\vee(o_3 \in \overline{is}\text{-}name\text{-}of\text{-}domain(p) \wedge \textbf{DEPENDENCE-DOMAIN}(o_3,o_2,o))))\Big)$$

There must be no interdependencies of the domains (recursive declarations of domain).

$$\textbf{NONREC-DOMAIN-DECLARATION}(o) \sim$$
$$\forall o_1 \forall o_2 \big(o_1,o_2 \in \overline{is}\text{-}name\text{-}of\text{-}domain(o) \wedge o_1 \neq o_2$$
$$\equiv \neg (\textbf{DEPENDENCE-DOMAIN}(o_1,o_2,o) \wedge \textbf{DEPENDENCE-DOMAIN}(o_2,o_1,o))\big)$$

◊ Checking declaration's being and being unique **EXIST-UNIQ-DECLARATION**(o).

In each part any *name* (except *name-of index iteration*) must be declared.

$$\textbf{EXIST-UNIQ-DECLARATION}(o) \sim$$
$$\forall p\Big(\quad p \in \overline{is}\text{-}name\text{-}of\ domain(o) \equiv \exists! q(q \propto \overline{is}\text{-}declaration\text{-}of\text{-}domain(o)$$
$$\wedge \textbf{NAME-DOMAIN-DECLARE}(p,q))$$
$$\vee p \in \overline{is}\text{-}name\text{-}of\ scalar(o) \equiv$$
$$\exists! q \exists! r(q \in \overline{is}\text{-}name\text{-}of\ scalar(o)$$
$$\wedge r \in \overline{is}\text{-}declaration\text{-}of\text{-}scalar\text{-}variables(o) \wedge q \propto r \wedge p = q)$$
$$\vee p \in \overline{is}\text{-}name\text{-}of\text{-}variable\text{-}on\text{-}domain(o) \equiv$$
$$\exists! q \exists! r(q \in \overline{is}\text{-}name\text{-}of\text{-}variable\text{-}on\text{-}domain(o)$$
$$\wedge r \in \overline{is}\text{-}declaration\text{-}of\text{-}variables\text{-}on\text{-}domains(o) \wedge q \propto r \wedge p = q)$$
$$\vee p \in \overline{is}\text{-}name\text{-}of\text{-}index\text{-}construction(o) \equiv$$
$$\exists! q \exists! r(q \in \overline{is}\text{-}name\text{-}of\text{-}index\text{-}construction(o)$$
$$\wedge r \in \overline{is}\text{-}declaration\text{-}of\text{-}index\text{-}construction(o) \wedge q \propto r \wedge p = q)$$
$$\vee p \in \overline{is}\text{-}name\text{-}of\text{-}domain's\text{-}parameters(o) \equiv$$
$$\exists! q \exists! r(q \in \overline{is}\text{-}name\text{-}of\text{-}domain's\text{-}parameters(o)$$
$$\wedge r \in \overline{is}\text{-}declaration\text{-}of\text{-}domain's\text{-}parameters(o) \wedge q \propto r \wedge p = q)$$
$$\vee p \in \overline{is}\text{-}name\text{-}of\text{-}external\text{-}function(o) \equiv$$
$$\exists! q \exists! r(q \in \overline{is}\text{-}name\text{-}of\text{-}external\text{-}function(o)$$
$$\wedge r \in \overline{is}\text{-}declaration\text{-}of\text{-}external\text{-}functions(o) \wedge q \propto r \wedge p = q)$$
$$\vee p \in \overline{is}\text{-}name\text{-}of\text{-}external\text{-}simple\text{-}part(o) \equiv$$

$$\exists!q\exists!r(q \in \overline{is}\text{-}name\text{-}of\text{-}external\text{-}parts(o)$$
$$\wedge r \in \overline{is}\text{-}declaration\text{-}of\text{-}external\text{-}simple\text{-}parts(o) \wedge q \propto r \wedge p = q)$$
$$\vee p \in \overline{is}\text{-}name\text{-}of\text{-}index(o) \equiv$$
$$\exists q\exists r(q \in \overline{is}\text{-}name\text{-}of\text{-}index(o)$$
$$\wedge r \in \overline{is}\text{-}one\text{-}dimensional\text{-}domain(o) \wedge q \propto r \wedge p = q))$$

Function checking *name-of-domain* declaring in *declaration-of-domain q*

$$\textbf{NAME-DOMAIN-DECLARE}(p,q) \sim$$
$$\exists s(r \in \overline{is}\text{-}s(q) \wedge q = p \bullet : \bullet r)$$
$$\vee \exists s_1 \exists s_2 (r \in \overline{is}\text{-}s_1(q) \wedge u \in \overline{is}\text{-}s_2(q) \wedge (q = p \bullet , \bullet u \bullet : \bullet r \vee q = u \bullet , \bullet p \bullet : \bullet r))$$

□ *Eliminating macroindices* **MACRO-INDEX-EXCLUDE**(*o*).

Any *name-of-index-construction* included into *index expressions* is substituted to corresponding, **list**-*explicit-id-expression* which is declared in *declaration-of-index-construction*. Then *declaration-of index-construction* is eliminated from the text of the program.

$$\textbf{MACRO-INDEX-EXCLUDE}(o) \sim \quad (\forall p(p \in \overline{is}\text{-}declaration\text{-}of\text{-}index\text{-}construction(o)$$
$$\wedge q \in \overline{is}\text{-}name\text{-}of\text{-}index\text{-}construction(p)$$
$$\wedge r \in \overline{is}\text{-}\textbf{list}\text{-}explicit\text{-}ind\text{-}expression(p))$$
$$\rightarrow ReplaceAll(q,r,o); Delete(p,o))$$

□ *Constructing index space* **DEF-INDEX-SPACE**(*o*).

The order of directions (of axes) of index space coincides with the arrangement of the index names in *declaration-of-domains'-indices* (from the left to the right). If there is no *declaration-of-domains'-indices* then the order of the directions is defined by the arrangement of *names-of-indices* in the text of the program.

$$\textbf{DEF-IND-SPACE}(o) \sim$$
$$T := \textbf{GET-PART-NAME}(o);$$
$$(\forall p(p \in \overline{is}\text{-}declaration\text{-}of\text{-}domains'\text{-}indices(o) \wedge q \in \overline{is}\text{-}\textbf{list}\text{-}name\text{-}of\text{-}index(p))$$
$$\rightarrow ReplaceAll(,,\bullet,q); AddListIndex(q,T); Delete(p,o);$$
$$\forall q(q \in \overline{is}\text{-}name\text{-}of\text{-}index(o))$$
$$\rightarrow AddIndex(q,T));$$
$$IndSpace := IndSpace \cup \{T\};$$

Function **GET-PARTNAME**(*o*) give the name of part *o*

$$\textbf{GET-PARTNAME}(o) \sim$$
$$(is\text{-}main\text{-}part(o) \wedge \exists n(is\text{-}name\text{-}of\text{-}main\text{-}part(n) \wedge \textbf{MAIN PART} \bullet n \propto o)$$
$$\rightarrow n;$$
$$(is\text{-}simple\text{-}part(o) \wedge \exists n(is\text{-}name\text{-}of\text{-}simple\text{-}part(n) \wedge \textbf{PART} \bullet n \propto o)$$
$$\rightarrow n;$$
$$(is\text{-}part\text{-}function(o) \wedge \exists n(is\text{-}name\text{-}of\text{-}part\text{-}function(n) \wedge \textbf{FUNCTION} \bullet n \propto o)$$
$$\rightarrow n;$$

Operator $AddListIndex(q,T)$ puts list of indices q into index space T and if relation $q \prec_o r$ is true then $q \prec_T r$ where $r \in \overline{is}$ - list - $name$ - of - $index(o)$.

Operator $AddIndex(q,T)$ puts name of index q into index space T (if it hasn't been there yet) and if relation $q \prec_o r$ is true then $q \prec_T r$ where $r \in \overline{is}$ - $name$ - of - $index(o)$ for o.

□ *Substituting domains' parameters* **PAR-DOMAIN-EXCLUDE**(o).

The *values of domains' parameters* are substituted instead of the *names of domains' parameters* in the text of the part.

$$\textbf{PAR-DOMAIN-EXCLUDE}(o) \sim$$
$$(\forall p(p \in \overline{is} - declaration - of - domains' - parameters(o)$$
$$\wedge q \in \overline{is} - name - of - domain's - parameter(p)$$
$$\wedge r \in \overline{is} - integer - without - sign(p) \wedge q \bullet = \bullet r \propto p)$$
$$\rightarrow ReplaceAll(q,r,o); Delete(p,o))$$

□ *Calculating constant expressions* **CONST-EXPR-EVALUATE**(o).

The values of all the constant expressions are computed.

$$\textbf{CONST-EXPR-EVALUATE}(o) \sim (\forall p(p \in \overline{is} - constant - expression(o))$$
$$\rightarrow \textbf{EVALUATE-EXPR}\ (p)$$
$$\forall p(p \in \overline{is} - constant - expression - without - sign(o))$$
$$\rightarrow \textbf{EVALUATE-EXPR}\ (p);)$$

Function **EVALUATE-EXPR** (e) computes the value of expression. Semantics of computing the expression has traditional meaning and may be defined by a well-known algorithm of translating arithmetical expression into Polish notation [12] and the rules of this notation's interpretation.

□ *Making domains standard* **STANDARD-DOMAIN**(o).

The main purpose of domains' standardisation is to reduce all the declarations of rectangular domains to the form

declaration-of-rectangular-domain:
    multidimensional-domain
multidimensional-domain:
    one-dimensional-domain
    name-of-one-dimensional-domain: (*domain-product*)
domain-product:
    component-domain { ; *component-domain* }⁻
component-domain:
    one-dimensional-domain
    (*domain-product*)
one-dimensional-domain:
    name-of-one-dimensional-domain: (*name-of-index*=*value*)

STANDARD-DOMAIN(o) ~ **IMPLICIT-NAMES**(o);
DEF-DOMAIN-EXCLUDE(o);

NAME-DOMAIN-REPLACE(o);
RECT-N-DOMAIN-TRANSFORM(o)

Function **IMPLICIT-NAMES**(o) carries out the following transformation: unnamed *new-domains-without-name* defined in operators **ASSUME, INPUT, OUTPUT** are substituted in these operators to the unique names, declarations of these domains are included into the program under corresponding names.

IMPLICIT-NAMES(o) ~

$$(\forall p((p \in \overline{is}\text{-}operator\text{-}\mathbf{ASSUME}(o)$$
$$\vee p \in \overline{is}\text{-}declaration\text{-}of\text{-}input(o)$$
$$\vee p \in \overline{is}\text{-}declaration\text{-}of\text{-}output(o))$$
$$\wedge \exists q(q \in \overline{is}\text{-}new\text{-}domain\text{-}without\text{-}name(p)))$$
$$\rightarrow name := uname(P); Replace(q, name, p); Include(name \bullet\!:\!\bullet\ q, o))$$

Function **DEF-DOMAIN-EXCLUDE**(o) carries out the following transformation: declaration def_1 of every *multidimensional-domain* with name *name-of-multidimensional-domain* and every *one-dimensional domain* with name *name-of-one-dimensional-domain* used as the component of declaration def_2 of another *multidimensional-domain* are included as the addition into the text of part o. *Name-of-multidimensional-domain*: or *name-of-one-dimensional-domain* included into def_1 is eliminated from def_2.

DEF-DOMAIN-EXCLUDE (o) ~

$$(\forall p(\ \ p \in \overline{is}\text{-}declaration\text{-}of\text{-}rectangular\text{-}domain(o)$$
$$\wedge(q \in \overline{is}\text{-}name\text{-}of\text{-}multidimensional\text{-}domain(p)$$
$$\vee q \in \overline{is}\text{-}name\text{-}of\text{-}one\text{-}dimensional\text{-}domain(p))$$
$$\wedge \exists s_1 \exists s_2 \exists s_3 (r_1 \in \overline{is}\text{-}s_1(p) \wedge r_2 \in \overline{is}\text{-}s_2(p) \wedge r_3 \in \overline{is}\text{-}s_3(p)$$
$$\wedge p = r_1 q \bullet\!:\!(\bullet r_2 \bullet) \bullet r_3))$$
$$\rightarrow Replace(q \bullet\!:\!(\bullet r_2 \bullet), (\bullet r_2 \bullet), p); Include(q \bullet\!:\!(\bullet r_2 \bullet), o))$$

Function **NAME-DOMAIN-REPLACE**(o) carries out the following transformation: every name-of-multidimensional-domain and every name of one-dimensional-domain used as the component of declaration def_2 of another multidimensional-domain is substituted to corresponding to these names declaration of multidimensional-domain or one-dimensional-domain in declaration def_2.

NAME-DOMAIN-REPLACE(o) ~

$$(\forall p \forall q(\ \ p \in \overline{is}\text{-}declaration\text{-}of\text{-}rectangular\text{-}domain(o)$$
$$\wedge(q \in \overline{is}\text{-}name\text{-}of\text{-}multidimensional\text{-}domain(p)$$
$$\vee q \in \overline{is}\text{-}name\text{-}of\text{-}one\text{-}dimensional\text{-}domain(p))$$
$$\wedge(q \bullet;\propto p \vee q \bullet) \propto p)$$
$$\wedge \exists u\ (\mathbf{NAME\text{-}DOMAIN\text{-}DECLARE}(q,u) \wedge u = q \bullet\!:\!\bullet v))$$
$$\rightarrow Replace(q,v,p))$$

Function **RECT-N-DOMAIN-TRANSFORM**(o) transforms the declarations of rectangular *new* domains into equivalent declarations of rectangular *multidimensional* domains.

RECT-N-DOMAIN-TRANSFORM(o) ~

$(\forall p($ $p \in \overline{is}\text{-}declaration\text{-}of\text{-}rectangular\text{-}domain(o)$

$\wedge is\text{-}new\text{-}domain(p)$

$\wedge \exists u \exists v(u \in \overline{is}\text{-}name\text{-}of\text{-}rectangular\text{-}domain(p)$

$\wedge v \in \overline{is}\text{-}\mathbf{list}\text{-}modification(p) \wedge p = q \bullet : \bullet u \bullet / \bullet v)$

$\rightarrow (is\text{-}name\text{-}of\text{-}one\text{-}dimensional\text{-}domain(u) \wedge \exists w$ (**NAME-DOMAIN-DECLARE**(u, w))

$\rightarrow Replace(q \bullet : \bullet u \bullet / \bullet v, q \bullet : \bullet$ **MODIFY**(w,v,o), p);

$is\text{-}name\text{-}of\text{-}multi\text{-}dimensional\text{-}domain(u) \wedge \exists w$ (**NAME-DOMAIN-DECLARE**(u, w))

$\exists s(s \in \overline{is}\text{-}name\text{-}new\text{-}domain(w))$

$\rightarrow$ **NAME-DOMAIN-REPLACE**(o);

$is\text{-}name\text{-}of\text{-}multi\text{-}dimensional\text{-}domain(u) \wedge \exists w$ (**NAME-DOMAIN-DECLARE**(u, w))

$\rightarrow Replace(q \bullet : \bullet u \bullet / \bullet v, q \bullet : \bullet$ **MODIFY**(w,v,o), p)));

NAME-DOMAIN-REPLACE(o)

Function **MODIFY**(w,v,o) chooses the next element from the list of modifications v.

MODIFY(w,v,o) ~

($is\text{-}modification(v) \rightarrow$ **MDFONE**(w,v,o);

$is\text{-}\mathbf{list}\text{-}modification(v) \rightarrow$ **MDFONE**(w,$head(v)$,o); **MODIFY**(w,$tail(v)$,o))

Functions **MDFONE**(w,v,o) and **CORRECT**(n,g) correct the domain according to modification v.

MDFONE(w,v,o) ~

$(\exists m \exists n(m \in \overline{is}\text{-}name\text{-}index(v) \wedge n \in \overline{is}\text{-}value(v) \wedge v{=}m \bullet = \bullet n$

$\wedge \exists a(a \in \overline{is}\text{-}value(w) \wedge w{=}m \bullet = \bullet a)$

$\rightarrow Replace(m \bullet = \bullet a, m \bullet = \bullet n, w);$

$\exists m \exists g(m \in \overline{is}\text{-}name\text{-}of\text{-}one\text{-}dimensional\text{-}domain(v) \wedge g \neq \varnothing \wedge v = m \bullet g$

$\wedge \exists c \exists d(c \in \overline{is}\text{-}name\text{-}index(o) \wedge d \in \overline{is}\text{-}value(o) \wedge m \bullet : (\bullet c \bullet = \bullet d)$

$\rightarrow$ **MDFONE**(w, $c \bullet = \bullet$ **CORRECT**(d,g)))

CORRECT(n,g) ~

$(\exists a \exists b(is\text{-}\text{int-}\text{constant}(a) \wedge is\text{-}\text{int-}\text{constant}(b) \wedge n = a \bullet .. \bullet b)$

$\rightarrow (\exists c(is\text{-}\text{int-}\text{constant}(c) \wedge g = +\mathbf{LEFT}(\bullet c \bullet) \rightarrow \text{evaluate-expr}(a - c) \bullet .. \bullet b)$

$\exists c(is\text{-}\text{int-}\text{constant}(c) \wedge g = -\mathbf{LEFT}(\bullet c \bullet) \rightarrow \text{evaluate-expr}(a + c) \bullet .. \bullet b)$

$\exists c(is\text{-}\text{int-}\text{constant}(c) \wedge g = -\mathbf{RIGHT}(\bullet c \bullet) \rightarrow a \bullet .. \bullet \text{evaluate-expr}(b - c))$

$$\exists c(is\text{-}\mathrm{int}\text{-}\mathrm{constant}(c) \wedge g = +\mathbf{RIGHT}(\bullet c \bullet) \rightarrow a \bullet .. \bullet \mathrm{evaluate}\text{-}\mathrm{expr}(b + c)))$$

□ *Making* **ASSUME** *operators standard* standard-**ASSUME**(o).

Every operator **ASSUME** consists of several relations is transformed into equivalent sequence of operators where each operator **ASSUME** consists of a relation.

standard-**ASSUME**(o) ~

$$(\forall p(p \in \overline{is}\text{-}operator\text{-}\mathbf{ASSUME}(o) \wedge q \in \overline{is}\text{-}name\text{-}domain(p)$$
$$\wedge \mathbf{FOR} \bullet q \bullet \mathbf{ASSUME} \propto p)$$
$$\rightarrow ReplaceAll(;,.\mathbf{FOR} \bullet q \bullet \mathbf{ASSUME}, p))$$

2.6 Definition of translator $Tr(P_s)$

Translator Tr carries out preparation of abstract program $A(P)$ for AM machine on standard representation P_s or fixes semantic errors.

$$
\begin{aligned}
Tr(P_s) \sim \big(&\forall o(o \in \overline{is}\text{-}part(P)) \rightarrow \\
&\quad \textbf{RECT-M-DOMAIN-EVALUATE}(o); \\
&\quad \textbf{DIAG-DOMAIN-EVALUATE}(o); \\
&\quad \textbf{VAR-DECLARATION}(o); \\
&\quad \textbf{REBUILD-INDEX}(o); \\
&\quad \textbf{COND-DOMAIN-EVALUATE}(o); \\
&\quad \textbf{INPUT-DECLARATION}(o); \\
&\quad \textbf{OUTPUT-DECLARATION}(o)); \\
&\textbf{INPUT-DATA-PROCESSING}(Files); \\
&\textbf{CHECK-INPUT-DATA}(P); \\
&\textbf{CALL-GRAPH-CREATE}(P); \\
&\textbf{CHECK-CALL-GRAPH}(P); \\
&\big(\forall o(o \in \overline{is}\text{-}part(P)) \rightarrow \\
&\quad \textbf{OPERATORS-TO-RELATIONS}(o)\big)
\end{aligned}
$$

2.7 Specification of translator's operators

Computing rectangular multidimensional domains **RECT-M-DOMAIN-EVALUATE**(o)

Declarations of rectangular *multidimensional* domains are transformed into the tuples of relation *Domains*. The order of indices in domains' declaration is determined by the indexes' order set in *Indspace*. Coordinates of the points from index space making up rectangular multidimensional domain are calculated statically.

Function **RECT-M-DOMAIN-EVALUATE** uses supplementary structures:

$DomainNotOrdered(q) = q \circ (u_1, v_1) \circ \cdots \circ (u_n, v_n)$, where (6)

q - *name-of-domain*,

u_i, v_i - *name-of-index* and its value,

n - dimension of domain;

$DomainOrdered(q) = q \circ Iname \circ v_1 \circ \cdots \circ v_n$, where (7)

q - *name-of-domain*,

u_i, v_i - *name-of-index* and its value, $Iname = u_1 \circ \cdots \circ u_n$, where if $u_i \prec_{Indspace(o)}$, then $u_i \prec_{DomainOrdered(q)} u_j$;

n - dimension of domain.

$$
\begin{aligned}
&\textbf{RECT-M-DOMAIN-EVALUATE}(o) \sim \\
&\big(\forall p(p \in \overline{is}\text{-}declaration\text{-}of\text{-}rectangular\text{-}domain(o) \\
&\quad \wedge is\text{-}multidimensional\text{-}domain(p) \\
&\quad \wedge q \in \overline{is}\text{-}name\text{-}of\text{-}domain(p)) \\
&\qquad \rightarrow DomainNotOrdered(q) := q; \\
&\qquad (\forall u \forall v(u \in \overline{is}\text{-}name\text{-}of\text{-}index(p) \wedge v \in \overline{is}\text{-}value(p) \wedge u \bullet = \bullet v \propto p) \\
&\qquad\quad \rightarrow DomainNotOrdered(q) := DomainNotOdered(q) \circ (u, v));
\end{aligned}
$$

ORDER-INDEX (*DomainNotOrdered*(*q*),*IndSpace*(*o*),*DomainOrdered*(*q*));
MAKE-DOMAIN (*DomainOrdered*(*q*),*R*);
$T \otimes R$ **ADD-TO** *Domains*;
Delete(*p*,*o*))

Function **ORDER-INDEX** (*DomainNotOdered*(*q*), *IndSpace*(*o*), *DomainOdered*(*q*)) transforms representation (6) into (7).

Function **MAKE-DOMAIN** (*DomainOdered*(*q*),*R*) forms auxiliary 'basic" relation R for domain q on representation (7). This relation differs from *Domains* by absence of attribute *Cond*.

MAKE-DOMAIN (*DomainOdered*(*q*),*R*) ~
$R := q \otimes Iname(q) \otimes$ **GENERATE**$(v_1) \otimes \cdots \otimes$ **GENERATE**(v_n)
GENERATE $(u \bullet \cdots \bullet v) \sim$ **F**$(\prod(I), u \le I \le v)$

□ *Computing diagonal domains* **DIAG-DOMAIN-EVALUATE**(*o*).

Declarations of *diagonal* domains are transformed into the tuples of relation *Domains*. The order of indices in domains' declaration is determined by the order of indices set in *IndSpace*. Coordinates of the points from index space formed new rectangular domain are computed statically.

Function **DIAG-DOMAIN-EVALUATE** uses the representation in *Domains for unconditional-domain p* obtained earlier. Then unconditional domain p is modified into *diagonal-domain*.

Differed from functions **RECT-M-DOMAIN-EVALUATE** and **RECT-N-DOMAIN-EVALUATE** function **DIAG-DOMAIN-EVALUATE** puts only those values of indices from domain p into relation *Domains* which satisfy *conditions-on-indices* set in declaration of diagonal domain.

DIAG-DOMAIN-EVALUATE(*o*) ~
$(\forall p(p \in \overline{is}\text{-}declaration\text{-}of\text{-}diagonal\text{-}domain(o)$
$\wedge \exists u \exists v \exists q(\overline{is}\text{-}name\text{-}of\text{-}unconditional\text{-}domain(p)$
$\wedge q \in \overline{is}\text{-}name\text{-}of\text{-}diagonal\text{-}domain(p))$
$\wedge v \in \overline{is}\text{-}\mathbf{list}\text{-}condition\text{-}on\text{-}domain(p)$
$\wedge p = q \bullet : \bullet u \bullet / \bullet v))$
$\rightarrow R := \mathbf{F}(Domains(\overline{Cond}), D = u);$
$T \otimes q \otimes \mathbf{F}(R, ReplaceAll(,,\wedge,v))$ **ADD-TO** *Domains;*
Delete(*p*,*o*))

□ *Declaring the variables* **VAR-DECLARATION** (*o*).

Declaration of *scalar variables* and *variables on domain* are transformed into tuples of relation *Variables*. For every *variables on domain* mentioned in the list of *iterated variables name of iteration index* is indicated as attribute *Iter*.

VAR-DECLARATION (*o*) ~
$(\forall p(p \in \overline{is}\text{-}declaration\text{-}of\text{-}scalar\text{-}variables(o)$
$\wedge \forall u(u \in \overline{is}\text{-}name\text{-}of\text{-}scalar(p)))$
$\rightarrow (\forall q(q \in \overline{is}\text{-}header\text{-}of\text{-}iteration(o)$

$$\wedge\exists\, r(r \in \overline{is}\text{-}name\text{-}\underline{of\text{-}iteration\text{-}index}(q)$$
$$\wedge\, u \propto q \rightarrow \varnothing \circ u \circ r \quad \textbf{ADD-TO}\ \ Variables; Delete(p,o);$$
$$\rightarrow \varnothing \circ u \circ \Lambda \quad \textbf{ADD-TO}\ \ Variables; Delete(p,o));$$
$$\forall p(p \in \overline{is}\text{-}declaration\text{-}of\text{-}variables\text{-}on\text{-}domain(p)$$
$$\wedge\forall u(u \in \overline{is}\text{-}declaration\text{-}of\text{-}variables\text{-}on\text{-}domain(p)$$
$$\wedge\forall v\exists w(v \in \overline{is}\text{-}name\text{-}\underline{of\text{-}variable\text{-}on\text{-}domain}(u) \wedge w \in \overline{is}\text{-}name\text{-}\underline{of\text{-}domain}(u))))$$
$$\rightarrow (\forall q(q \in \overline{is}\text{-}header\text{-}of\text{-}iteration(o)$$
$$\wedge\exists\, r(r \in \overline{is}\text{-}name\text{-}\underline{of\text{-}iteration\text{-}index}(q)$$
$$\wedge v \propto q \rightarrow w \circ v \circ r \quad \textbf{ADD-TO}\ \ Variables; Delete(p,o);$$
$$\rightarrow w \circ v \circ \Lambda \quad \textbf{ADD-TO}\ \ Variables; Delete(p,o)))$$

□ *Restoring indices* **REBUILD-INDEX**(o)

In using *variable on domain* the default rule of setting index is right (*index expressions* coincided with *the name of index* may be omitted). Function **REBUILD-INDEX**(o) restores the indices set default in the index expressions of the variables declared on domains based on the information about these domains. The indices indicated as the formal parameters of the parts or function, actual parameters set on domain, elements of the list of iterated variables aren't to be restored .

REBUILD-INDEX$(o) \sim$
$$(\forall p(p \in \overline{is}\text{-}name\text{-}\underline{of\text{-}variable\text{-}on\text{-}domain}(o)$$
$$\wedge q \in \overline{is}\text{-}\textbf{list}\text{-}\underline{ind\text{-}expression}(o)$$
$$\forall p \bullet [\bullet q \bullet] \propto o)$$
$$\rightarrow ReplaceAll(p \bullet [\bullet q \bullet], p \bullet (\bullet \textbf{REBUILD}(q, \textbf{GET-INDEX}(p)) \bullet), o);$$
$$\forall p(p \in \overline{is}\text{-}name\text{-}\underline{of\text{-}variable\text{-}on\text{-}domain}(o)$$
$$\wedge\neg\exists\, q((q \in \overline{is}\text{-}formal\text{-}parameter\text{-}of\text{-}part(o) \wedge p \propto q)$$
$$\vee (q \in \overline{is}\text{-}formal\text{-}parameter\text{-}of\text{-}function(o) \wedge p \propto q)$$
$$\vee (q \in \overline{is}\text{-}initial\text{-}parameter(o) \wedge p \bullet \textbf{ON} \propto q)$$
$$\vee (q \in \overline{is}\text{-}parameter\text{-}result(o) \wedge p \bullet \textbf{ON} \propto q)$$
$$\vee (q \in \overline{is}\text{-}iterated\text{-}element(o) \wedge p \propto q))$$
$$\rightarrow ReplaceAll(p, p \bullet (\bullet ReplaceAll(\circ,,, \textbf{GET-INDEX}(p)) \bullet), o))$$

Auxiliary function **GET-INDEX**(y) gives indices of variable y in accordance with the definition domain of this variable.

GET-INDEX$(y) \sim R_1 := \textbf{F}(Variables(D, X, Iter), X = y);$
$$R_2 := \textbf{F}(Domains(Cond, D, Iname, I), D = R_1(D));$$
$$(R_1(Iter) = \varnothing\varnothing \rightarrow R_2(Iname);$$
$$R_1(Iter) \neq \varnothing\varnothing \rightarrow R_2(Iname) \circ R_1(Iter))$$

Auxiliary function **REBUILD**$(q,Iname)$ changes the list of index expressions q to $Iname$ substituting index expressions set explicitly from q to $Iname$ instead of corresponding indices (substitution is done by function **CHANGE**).

$$\textbf{REBUILD}(q,Iname) \sim ReplaceAll(\circ,,,Iname);$$
$$\forall p(p \in \overline{is}\text{-}ind\text{-}expression(q) \rightarrow \textbf{CHANGE}(Iname,p))$$

$$\textbf{CHANGE}(Iname,p) \sim$$
$$(\exists u \exists v \exists w(is\text{-}name\text{-}\underline{of\text{-}index}(u) \wedge is\text{-}name\text{-}\underline{of\text{-}index}(v)$$
$$\wedge is\text{-}\mathrm{int}\text{-}\mathrm{constant}(w) \wedge p = u \bullet = \bullet v \bullet + \bullet w$$
$$\rightarrow Replace(u, v \bullet + \bullet w, Iname);$$
$$\exists u \exists v \exists w(is\text{-}name\text{-}\underline{of\text{-}index}(u) \wedge is\text{-}name\text{-}\underline{of\text{-}index}(v)$$
$$\wedge is\text{-}\mathrm{int}\text{-}\mathrm{constant}(w) \wedge p = u \bullet = \bullet v \bullet - \bullet w$$
$$\rightarrow Replace(u, v \bullet - \bullet w, Iname);$$
$$\exists u \exists v \exists w(is\text{-}name\text{-}\underline{of\text{-}index}(u) \wedge is\text{-}name\text{-}\underline{of\text{-}index}(v) \wedge p = u \bullet = \bullet v$$
$$\rightarrow Replace(u, v, Iname);$$
$$\exists u \exists v \exists w(is\text{-}name\text{-}\underline{of\text{-}index}(u) \wedge is\text{-}\mathrm{int}\text{-}\mathrm{constant}(w) \wedge p = u \bullet = \bullet v$$
$$\rightarrow Replace(u, v, Iname);$$
$$\exists u \exists v \exists w(is\text{-}name\text{-}\underline{of\text{-}index}(u) \wedge is\text{-}\mathrm{int}\text{-}\mathrm{constant}(w) \wedge p = u \bullet = \bullet v \bullet - \bullet w$$
$$\rightarrow Replace(u, v \bullet - \bullet w, Iname);$$
$$\exists u \exists v \exists w(is\text{-}name\text{-}\underline{of\text{-}index}(u) \wedge is\text{-}\mathrm{int}\text{-}\mathrm{constant}(w) \wedge p = u \bullet = \bullet v \bullet + \bullet w$$
$$\rightarrow Replace(u, v \bullet + \bullet w, Iname))$$

□*Computing conditional domains* **COND-DOMAIN-EVALUATE**(o).

Declarations of *conditional* domains are transformed into the tuples of relation *Domains*. Coordinates of the points from index space forming conditional domain are computed dynamically in the process of *AM* machine working at the expense of interpretation.

$$\textbf{COND-DOMAIN-EVALUATE}(o) \sim$$
$$(\forall p(p \in \overline{is}\text{-}declaration\text{-}of\text{-}conditional\text{-}domain(o)$$
$$\wedge \exists u \exists v \exists w \exists l(u \in \overline{is}\text{-}name\text{-}\underline{of\text{-}conditional\text{-}domain}(p)$$
$$\wedge v \in \overline{is}\text{-}name\text{-}\underline{of\text{-}conditional\text{-}domain}(p)$$
$$\wedge w \in \overline{is}\text{-}name\text{-}\underline{of\text{-}domain}(p)$$
$$\wedge l \in \overline{is}\text{-}condition\text{-}on\text{-}domain(p)$$
$$\wedge p = u \bullet , \bullet v \bullet : \bullet w \bullet l \bullet l))$$
$$\rightarrow RIname := \mathbf{F}(Domains, D = w)(Iname);$$
$$RI := \mathbf{F}(Domains, D = w)(I);$$
$$RCond := \mathbf{F}(Domains, D = w)(Cond);$$
$$RYes := \textbf{REPLACE-IND-VAL}(I), RIname, RI);$$
$$RNo := \textbf{REPLACE-IND-VAL}(\neg(I), RIname, RI);$$
$$RCond \circ RYes \otimes u \otimes RIname \circ RI \ \textbf{ADD-TO}\ Domains;$$
$$RCond \circ RNo \otimes v \otimes RIname \circ RI \ \textbf{ADD-TO}\ Domains;$$

$Delete(p,o)$)

Function **REPLACE-IND-VAL**$(l,RIname,RI)$ constructs the relation, which number of tuples is equal to the number of the tuples in relation RI, and the tuples themselves are the result of substituting the values of indices I instead of their names $Iname$ into l (the order of substitution is determined by the order of the I attribute's values).

□ *Declaring inputs* **INPUT-DECLARATION**(o).

Declaration of *input scalars* and *inputs on domain* are transformed into tuples of relation *Input* and the tuples of relation *Relations*.

INPUT-DECLARATION(o) ~

n := get-partname(o);

$(\forall p(p \in \overline{is}\text{-}input\text{-}scalar(o) \wedge \exists u(is\text{-}name\text{-}\underline{of\text{-}scalar}(u) \wedge u \propto p))$

$\rightarrow$ v := **GET-IN-FILENAME**(p);

$u \circ \Lambda \circ v$ **ADD-TO** *Input*;

$n \circ \varnothing \circ \varnothing \circ \mathbf{T} \circ \mathbf{T} \circ u \circ \Lambda \circ$**INPUT** **ADD-TO** *Relations*;

$Delete(p,o)$;

$\forall p(p \in \overline{is}\text{-}inputs\text{-}on\text{-}domain(o)$

$\wedge \forall u \exists w(u \in \overline{is}\text{-}input\text{-}on\text{-}domain(p) \wedge w \in \overline{is}\text{-}name\text{-}\underline{of\text{-}domain}(p)))$

$\rightarrow$ v := **GET-IN-FILENAME**(p);

$RIname$:= **F**$(Domains, D = w)(Iname)$;

RI:= **F**$(Domains, D = w)(I)$;

$var := u \bullet (\bullet ReplaceAll(\circ,,,RIname) \bullet)$;

Var:= **REPLACE-IND-VAL**$(var, RIname, RI)$;

$Var \otimes \Lambda \otimes v$ **ADD-TO** *Input*;

$n \otimes \varnothing \otimes \varnothing \otimes \mathbf{T} \otimes \mathbf{T} \otimes Var \otimes \Lambda \otimes v$ **ADD-TO** *Relations*;

$Delete(p,o)$)

Auxiliary function **GET-IN-FILENAME**(p) gives the name of input file of input variable p.

GET-IN-FILENAME(p) ~ ($\exists v(v \in \overline{is}\text{-}name\text{-}\underline{of\text{-}file}(p)) \rightarrow v$;

$\rightarrow$ 'norma.dat')

Semantics of attributes *format* for input variables isn't described in the given specifications it is equal to the corresponding semantics of the FORTRAN language.

□ *Declaring outputs* **OUTPUT-DECLARATION**(o).

Declarations of output scalars and outputs on domain are transformed into the tuples of relation *Output*.

OUTPUT-DECLARATION(o) ~

$(\forall p(p \in \overline{is}\text{-}output\text{-}scalar(o) \wedge \exists u(is\text{-}name\text{-}\underline{of\text{-}scalar}(u) \wedge u \propto p))$

$\rightarrow$ v := **GET-OUT-FILENAME**(p);

$\mathbf{T} \circ u \circ \Lambda \circ v$ **ADD-TO** *Output*; $Delete(p,o)$;

$\forall p(p \in \overline{is}\text{-}outputs\text{-}on\text{-}domain(o)$

$$\wedge\forall u\exists\, w(u \in \overline{is}\text{-}output\text{-}on\text{-}domain(p) \wedge w \in \overline{is}\text{-}name\text{-}of\text{-}domain(p)))$$

$\rightarrow$ v := **GET-IN-FILENAME**(p);
$RIname$:= **F**$(Domains, D = w)(Iname)$;
RI:= **F**$(Domains, D = w)(I)$;
var:= $u \bullet (\bullet ReplaceAll(\circ,,,RIname) \bullet)$;
Var:= **REPLACE-IND-VAL**$(var, RIname, RI)$;
$RCond \circ Var \otimes \Lambda \otimes v$ **ADD-TO** $Output$;
$Delete(p,o)$)

Auxiliary function **GET-OUT-FILENAME**(p) gives the name of output file for variable p.

$$\textbf{GET-OUT-FILENAME}(p) \sim (\ \exists\, v(v \in \overline{is}\text{-}name\text{-}of\text{-}file(p)) \rightarrow v;$$
$$\rightarrow \text{'display'})$$

Semantics of the attributes for output variables in this specification isn't given as it deals with the rules of representing output variables in the external media and is similar to semantics of corresponding specifications of FORTRAN output format.

□ *Processing input data files* **INPUT-DATA-PROCESSING**$(Files)$.

The values of *input scalars* and *inputs on domain* represented in input files $Files$ are transformed into the tuples of relations $InputData$.

INPUT-DATA-PROCESSING$(Files) \sim$
$\forall p(p \in Files \wedge q \in \overline{is}\text{-}name\text{-}of\text{-}file(p) \wedge \forall r(r \in \overline{is}\text{-}element\text{-}of\text{-}input(p)))$
$\rightarrow (\exists\, v\exists\, u(v \in \overline{is}\text{-}name\text{-}of\text{-}scalar(r) \wedge u \in \overline{is}\text{-}arithm\text{-}constant(r) \wedge r = v \bullet = \bullet u)$
$\rightarrow v \circ u \circ q$ **ADD-TO** $InputData$;
$\exists\, v\exists\, u\exists\, w(v \in \overline{is}\text{-}name\text{-}of\text{-}variable\text{-}on\text{-}domain(r)$
$\wedge u \in \overline{is}\text{-}\textbf{list}\text{-}range\text{-}of\text{-}index(r)$
$\wedge w \in \overline{is}\text{-}data(r) \wedge r = v \bullet (\bullet u \bullet) = \bullet w)$
$\rightarrow Var$:= **MAKE-INDEX**(v,u); **GETNEXT**(w) **ADD-TO** $Value$;
$Var \circ Value \otimes q$ **ADD-TO** $InputData$))

Auxiliary function **MAKE-INDEX**(v,u) constructs the relation with each tuple equal to the name of input variable v with index-constants which values are defined by the list of u indexes' ranges. Let list $u = u_1,\ldots,u_n$, where $u_i = name\text{-}index_i \bullet = \bullet a_i \bullet .. \bullet b_i$.

MAKE-INDEX$(v,u) \sim$
$Temp$:= **GENERATE**$(a_1 \bullet .. \bullet b_1) \otimes ... \otimes$ **GENERATE**$(a_n \bullet .. \bullet b_n)$;
$Temp$:=$ReplaceAll(\circ,,,Temp)$;
$Temp$:=$v \bullet (\otimes\ Temp\ \otimes)$

Function **GETNEXT**(w) gives the next element from the list of data w.

◊ *Checking input data* **CHECK-INPUT-DATA**(P).

Data must be defined in the input files for every variable introduced during the work. I.e. tuple β is said to be in relation *InputData* for every tuple α in relation *Input*, when the value of β differs from the value of α only by the value of attribute *Value*. If this condition is satisfied the value of *Value* is taken from tuple β and put into tuple α.

CHECK-INPUT-DATA$(P) \sim$

$$(Input(\overline{Value}) - InputData(\overline{Value}) = \varnothing$$
$$\rightarrow \forall\alpha(\alpha \in InputData \wedge \alpha = \langle var\ value\ file\rangle \wedge \beta \in Input \wedge \beta = \langle var\ \Lambda\ file\rangle$$
$$\rightarrow \beta := \langle var\ value\ file\rangle);$$
$$\rightarrow \mathbf{F});$$

□ *Designing the graph of parts and functions' calls* **CALL-GRAPH-CREATE**(P).

Directed graph of parts and functions' calls $G=(V,E)$ consists of set of nodes V and set of arcs E. Set $V = \{v_i\}$ consists of the names of parts and functions' called in program P. If there is call v_j from part (function) v_i then arc (v_i,v_j) is included into E. Every node v_i contains supplementary information consists of the list of formal parameters of part v_i (it may be empty). If there is no declaration of part but there is its call then the error is fixed in designing the graph and **F** becomes the result of function **CALL-GRAPH-CREATE**(P).

◊ *Checking the graph of functions and parts' calls for being correct* **CHECK-CALL-GRAPH**(P).

The graph of parts and functions' calls is correct if it is acyclic with one node which has no re-entrant arcs (this node corresponds to the call of the main part of program P and further will be called *entry node*).

Graph of parts and functions' calls $G=(V,E)$ is checked for being correct.

□ *Translating the operators into relation Relations* **OPERATORS-TO-RELATIONS** (o).

Operators **ASSUME** are unrolled. Operator specifying some rule of computation **F** for the variable of the program in all the points D_i from computation domain D, $i = 1,...,\|D\|$ is represented in the form of $\|D\|$ operators. Each operator specifies the rule of computation **F** in particular point D_i in computation domain D.

If operator **ASSUME** is set in construction *iteration* then it provides with the special mark in the field of attribute $ItVal = ItId \bullet C$ when it is included into relation *Relation*.

The conditions set in the declaration of conditional computation domain D and the condition of the computation's arguments being defined are taken into consideration in putting in the value of attribute *Cond*. The value of variable X with index I is depended on some arguments, let's name them *Arg*, $Arg = X_1(I_1),...,X_k(I_k)$, $X_i \in \mathbf{T}$, $I_i \in \mathbf{N}$. Then logical function *Def*(*Arg*) is added to condition *Cond*. Function *Def*(*Arg*) will be true ($\mathbb{T}$) if all $X_i \in Arg$ have already defined (i.e. their value is the value of attribute $Value \neq \Lambda$) in other case it will be false ($\mathbb{F}$). Function *Def*(*Arg*) is a built-in function of *AM*- machine and is performed in the process of its work.

Call to the reduction functions and external functions included into arithmetical expressions are eliminated from these arithmetical expressions and are represented by new scalar operators included into the program.

First all the constructions iteration are processed and after them the operators which are not included into the iterations.

OPERATORS-TO-RELATIONS$(o) \sim n :=$ **GET-PARTNAME**(o);

$$(\forall q(q \in \overline{is}\text{-}iteration(o))$$

$\rightarrow$ **ITERATION-TO-RELATIONS**(n,q);*Delete*(q,o);

$\forall p(p \in \overline{is}\text{-}operator(o))$

$\rightarrow$ **OPERATOR-TO-RELATIONS**$(p,n,\varnothing,\varnothing)$;*Delete*$(q,o)$))

Here are auxiliary functions performing analysis of construction *iteration q.*

Function **ITERATION-STRUCTURE**(q,a,b,c,d,e,i) for iteration q gives value **T**, if a - *header of iteration,* b *- boundary values of iteration,* c *- initial values of iteration,* d *- body of iteration,* e*- exit condition of iteration,* i *- index of iteration*:

ITERATION-STRUCTURE$(q, a, b, c, d, e, i) \sim$

$\exists a(is\text{ - head - of - iteration}(a))$

$\wedge\ \exists b(is\text{ - boundary - values}(b))$

$\wedge\ \exists c(is\text{ - initial - values}(c))$

$\wedge\ \exists d(is\text{ - body - of - iteration}(d))$

$\wedge\ \exists e(is\text{ - exit - conditions}(e))$

$\wedge\ \exists i(is\text{ - }\underline{name\text{ - }index\text{ - }iteration}(i))$

$\wedge\ a \bullet b \bullet c \bullet d \bullet e \bullet$ **END ITERATION** $\bullet\, b = q$

Functions **GET-HEAD-ITERATION**(q), **GET-BOUNDARY**(q), **GET-INITIAL**(q), **GET-BODY**(q), **GET-EXIT**(q), **GET-ITERATION-INDEX**(q) for iteration q give correspondingly header, boundary values, initial values, body, exit condition, index.

GET-HEAD-ITERATION$(q) \sim$ (**ITERATION-STRUCTURE**$(q, a, b, c, d, e, i) \rightarrow a$)

GET-BOUNDARY$(q) \sim$ (**ITERATION-STRUCTURE**$(q, a, b, c, d, e, i) \rightarrow b$)

GET-INITIAL$(q) \sim$ (**ITERATION-STRUCTURE**$(q, a, b, c, d, e, i) \rightarrow c$)

GET-BODY$(q) \sim$ (**ITERATION-STRUCTURE**$(q, a, b, c, d, e, i) \rightarrow d$)

GET-EXIT$(q) \sim$ (**ITERATION-STRUCTURE**$(q, a, b, c, d, e, i) \rightarrow e$)

GET-ITERATION-INDEX$(q) \sim$ (**ITERATION-STRUCTURE**$(q, a, b, c, d, e, i) \rightarrow i$)

Here is the function processing construction *iteration q.*

ITERATION-TO-RELATIONS$(n, q) \sim$

$a :=$ **GET-HEAD-ITERATION**(q);

$b :=$ **GET-BOUNDARY**(q);

$c :=$ **GET-INITIAL**(q);

$d :=$ **GET-BODY**(q);

$e :=$ **GET-EXIT**(q);

$i :=$ **GET-ITERATION-INDEX**(q);

$(\forall p(p \in \overline{is}\text{ - }operator(b))$

$\rightarrow$ *Include*(p,c); *Include*(p,d); *Delete*(p,q));

$(\forall p(p \in \overline{is}\text{ - }element\text{ - }of\text{ - }initials(c)$

$\wedge\ \exists a(is\text{ - int - const}(a) \wedge$ **INITIAL** $\bullet\, i = \bullet\, a \bullet : \propto p$

$\wedge\ \forall r(r \propto \overline{is}\text{ - }operator(p)))$

$\rightarrow$ **OPERATOR-TO-RELATIONS**(r, n, i, a));

Delete(c, q);

$(\forall r(r \propto \overline{is}\text{-}operator(d)))$
$\rightarrow$ **OPERATOR-TO-RELATIONS**$(r, n, i, 1)$);
EXIT-TO-RELATIONS$(e, n, i, 1)$)

Function **OPERATOR-TO-RELATIONS**(p, n, i, a) puts the tuples into relation *Relation*. The form of the tuples is determined by the type of operator p set in the part with name n (i - index of iteration or $\varnothing$, a - the value of iteration index or $\varnothing$).

OPERATOR-TO-RELATIONS$(p, n, i, a) \sim$
(is-operator-**ASSUME**$(p) \rightarrow$**ADD-ASSUME**(p, n, i, a);
is-scalar-operator$(p) \rightarrow$**ADD-SCALAR**(p, n, i, a);
is-call-of-part$(p) \rightarrow$**ADD-COMPUTE**(p, n, i, a))

Function **ADD-ASSUME**(p, n, i, a) converts operator **ASSUME** p into the tuples of relation *Relations*. Operator **ASSUME** p is set in the part with name n (i - index of iteration or $\varnothing$, a - the value of iteration index or $\varnothing$).

ADD-ASSUME$(p, n, i, a) \sim$
$d :=$ **GET-DOMAIN-NAME**(p);
$Rcond :=$ **F**$(Domains, D = d)(Cond)$;
$RIname :=$ **F**$(Domains, D = d)(Iname)$;
$RI :=$ **F**$(Domains, D = d)(I)$;
$RUnroll :=$ **REPLACE-IND-VAL**$(p, RIname, RI)$;
$RUnroll :=$ EXCLUDE-FUNCTION$(RUnroll, n, i, a)$;
$RLeft :=$ GET-LEFT-PART$(RUnroll)$;
$RRight :=$ GET-RIGHT-PART$(RUnroll)$;
$RVar :=$ GET-VAR$(RLeft)$;
$(\exists z(is\text{-}\text{relation}(z) \wedge z \propto p) \rightarrow RFunc := RRight)$;
$\exists z(is\text{-}\text{call}\text{-}\text{of}\text{-}\text{part}(z) \wedge z \propto p) \rightarrow RFunc := RUnroll)$;
$RDep :=$ **Def**$(\circ$ GET-DEPENDENCE$(RRight)\circ)$;
$RLambda := RVar$; $(\forall c(c \in RLambda) \rightarrow$ **UPDATE**$(c, \Lambda, RVar))$;
$n \circ ia \circ RDep \circ RCond \circ RVar \circ RLambda \circ RFunc$ **ADD_TO** *Relations*

Function **ADD-COMPUTE**(p, n, i, a) converts operator **COMPUTE** p into the tuples of relation *Relations*. Operator **COMPUTE** p is set in the part with name n (i - index of iteration or $\varnothing$, a - the value of iteration index or $\varnothing$).

ADD-COMPUTE$(p, n, i, a) \sim$
$q :=$ EXCLUDE-FUNCTION(p, n, i, a);
$l :=$ **GET-LEFT-PART**(q);
$r :=$ **GET-RIGHT-PART**(q);
$RVar :=$ **GET-VAR**$(l, \varnothing, \varnothing)$;
$RFunc := q$;
$RDep :=$ **Def**$(\circ$ **GET-DEPENDENCE**$(r)\circ)$;
$RLambda := Rvar$; $(\forall c(c \in RLambda) \rightarrow$ **UPDATE**$(c, \Lambda, RVar))$;
$n \circ ia \circ RDep \circ \mathbf{T} \circ RVar \circ RLambda \circ RFunc$ **ADD_TO** *Relations*

Function **ADD-SCALAR**(p, n, i, a) converts scalar operator p into the tuples of relation *Relations*. Scalar operator p is set in the part with name n (i - index of iteration or $\varnothing$, a - the value of iteration index or $\varnothing$).

ADD-SCALAR(p, n, i, a) ~
q := **EXCLUDE-FUNCTION**(p, n, i, a);
l := **GET-LEFT-PART**(q);
r := **GET-RIGHT-PART**(q);
RDep := **Def**(∘ **GET-DEPENDENCE**(r) ∘);
$n \circ ia \circ RDep \circ \mathbf{T} \circ l \circ \Lambda \circ r$ **ADD_TO** *Relations*

Function **EXIT-TO-RELATIONS**(e, n, i, a) converts operator of iteration's exit condition e into the tuples of relation *Relations*. Operator of iteration's exit condition e is set in the part with name n (i - index of iteration or $\varnothing$, a - the value of iteration index or $\varnothing$).

EXIT-TO-RELATIONS(e, n, i, a) ~
q := **EXCLUDE-FUNCTION**(e, n, i, a);
RDep := **Def**(∘ **GET-DEPENDENCE**(q) ∘);
$n \circ ia \circ RDep \circ q \circ \varnothing \circ \varnothing \circ \varnothing$ **ADD_TO** *Relations*

Function **GET-DOMAIN-NAME**(p) gives the name to the domain of operator **ASSUME** p.

GET-DOMAIN-NAME(p) ~ $(\exists n(is\text{-}name\text{-}of\text{-}\underline{domain}(n) \wedge \mathbf{FOR} \bullet n \bullet \mathbf{ASSUME} \propto p \rightarrow n)$

Function **GET-LEFT-PART**(p) gives the left part of p operator's *relation* or the list of actual parameters-results placed after key-word **RESULTS** of *part's call* **COMPUTE** in operator p.

GET-LEFT-PART(p) ~
$(\exists n \exists m \exists s \exists t(is\text{-}name\text{-}of\text{-}\underline{domain}(n) \wedge is\text{-}name\text{-}of\text{-}\underline{simple\text{-}part}(m)$
$\wedge is\text{-list-}initial\text{-}parameter(s)$
$\wedge is\text{-list-}parameter\text{-}result(t)$
$\wedge \mathbf{FOR} \bullet n \bullet \mathbf{ASSUME}\ \ \mathbf{COMPUTE} \bullet m \bullet (\bullet s \bullet \mathbf{RESULTS} \bullet t \bullet) = p)$
$\rightarrow t;$
$\exists m \exists s \exists t(is\text{-}name\text{-}of\text{-}\underline{simple\text{-}part}(m)$
$\wedge is\text{-list-}initial\text{-}parameter(s)$
$\wedge is\text{-list-}parameter\text{-}result(t)$
$\wedge \mathbf{COMPUTE} \bullet m \bullet (\bullet s \bullet \mathbf{RESULTS} \bullet t \bullet) = p)$
$\rightarrow t;$
$\exists n \exists m \exists a(is\text{-}name\text{-}of\text{-}\underline{domain}(n) \wedge is\text{-arithm-expression}(a)$
$\wedge is\text{-variable-on-domain}(m)$
$\wedge \mathbf{FOR} \bullet n \bullet \mathbf{ASSUME} \bullet m \bullet = \bullet a = p)$
$\rightarrow m,$
$\exists m \exists a(is\text{-}arithm\text{-}expression(a) \wedge is\text{-}name\text{-}of\text{-}\underline{scalar}(m) \wedge m \bullet = \bullet a = p)$
$\rightarrow m)$

Function **GET-RIGHT-PART**(p) gives the right part of p operator's *relation* or the list of actual initial parameters placed before key-word **RESULTS** of *part's call* **COMPUTE** in operator p.

GET-RIGHT-PART(p) ~

$$
\begin{aligned}
(\ & \exists n \exists m \exists s \exists t (is\text{-}name\text{-}of\text{-}\underline{domain}(n) \wedge is\text{-}name\text{-}of\text{-}\underline{simple\text{-}part}(m) \\
& \quad \wedge is\text{-list-}initial\text{-}parameter(s) \\
& \quad \wedge is\text{-list-}parameter\text{-}result(t) \\
& \quad \wedge \mathbf{FOR} \bullet n \bullet \mathbf{ASSUME\ \ COMPUTE} \bullet m \bullet (\bullet s \bullet \mathbf{RESULTS} \bullet t \bullet) = p) \\
& \quad \rightarrow s; \\
& \exists m \exists s \exists t (is\text{-}name\text{-}of\text{-}\underline{simple\text{-}part}(m) \\
& \quad \wedge is\text{-list-}initial\text{-}parameter(s) \\
& \quad \wedge is\text{-list-}parameter\text{-}result(t) \\
& \quad \wedge \mathbf{COMPUTE} \bullet m \bullet (\bullet s \bullet \mathbf{RESULTS} \bullet t \bullet) = p) \\
& \quad \rightarrow s; \\
& \exists n \exists m \exists a (is\text{-}name\text{-}of\text{-}\underline{domain}(n) \wedge is\text{-arithm-expression}(a) \\
& \quad \wedge is\text{-variable-on-domain}(m) \\
& \quad \wedge \mathbf{FOR} \bullet n \bullet \mathbf{ASSUME} \bullet m \bullet = \bullet a = p) \\
& \quad \rightarrow a; \\
& \exists m \exists a (is\text{-}arithm\text{-}expression(a) \wedge is\text{-}name\text{-}of\text{-}\underline{scalar}(m) \wedge m \bullet = \bullet a = p) \\
& \quad \rightarrow a\ )
\end{aligned}
$$

Function **EXCLUDE-FUNCTION**(u, n, i, a) converts calls to the reduction functions and external functions set in operator u from the part with name n (i - index of iteration, a - the value of iteration index).

EXCLUDE-FUNCTION(u, n, i, a) ~

$$
\begin{aligned}
(\ & \forall f (f \in \overline{is}\text{-}call\text{-}to\text{-}reduction\text{-}function(u)) \\
& \quad \rightarrow t := uname(P); Replace(f, t, u); \mathbf{ADD\text{-}REDUCT\text{-}FUNC}(t, f, n, i, a); \\
& \forall f (f \in \overline{is}\text{-}call\text{-}to\text{-}external\text{-}function(u)) \\
& \quad \rightarrow t := uname(P); Replace(f, t, u); \mathbf{ADD\text{-}EXTERN\text{-}FUNC}(t, f, n, i, a))
\end{aligned}
$$

ADD-REDUCT-FUNC(t, v, n, i, a) ~

$$
\begin{aligned}
(\ & \exists q \exists s \exists a\ (is\text{-}name\text{-}\underline{reduction\text{-}function}(q) \wedge is\text{-}name\text{-}of\text{-}\underline{domain}(s) \\
& \wedge is\text{-}arithm\text{-}expression(a) \wedge q \bullet ((\bullet s \bullet) \bullet r \bullet) = v) \\
& \quad \rightarrow RIname := \mathbf{F}(Domains, D = s)(Iname); \\
& \quad\ \ RI := \mathbf{F}(Domains, D = s)(I); \\
& \quad\ \ SumUnroll := \mathbf{REPLACE\text{-}IND\text{-}VAL}(a, RIname, RI); \\
& \quad\ \ SumUnroll := \mathrm{EXCLUDE\text{-}FUNCTION}(SumUnroll, n, i, a); \\
& \quad\ \ RSumDep := \mathbf{Def}(\circ\, \mathrm{GET\text{-}DEPENDENCE}(SumUnroll) \circ); \\
& \quad\ \ SumDep := \mathbf{REL\text{-}IN\text{-}LIST}(RSumDep); \\
& \quad\ \ n \circ ia \circ SumDep \circ \mathbf{T} \circ t \circ \Lambda \circ v \quad \mathbf{ADD\_TO} \quad Relations\ )
\end{aligned}
$$

Auxiliary function **REL-IN-LIST**($RSumDep$) represents the tuples of relation $RSumDep$ in the form of the list of tuples enamurated by comma.

ADD-EXTERN-FUNC$(t, v, n, i, a) \sim$

$($ $\exists q \exists s(is\text{-}name\text{-}extern\text{-}function(q) \wedge is\text{-}$ list $\text{-}initial\text{-}parameter(s)$

$\wedge q \bullet (\bullet s \bullet) = v)$

$\rightarrow ExtDep :=$ **Def**$(\circ$ GET-DEPENDENCE$(s) \circ);$

$n \circ ia \circ ExtDep \circ \mathbf{T} \circ t \circ \Lambda \circ v$ **ADD_TO** *Relations* $)$

Function **GET-VAR**(l), *l - variable on domain* with indices-constants or **list**-*parameter-result* with indices constants creates the list of *variables on domain* with indices constants where all the variables set in l are included.

GET-VAR$(l) \sim$

$($ $is\text{-}name\text{-}of\text{-}scalar(l) \rightarrow l;$

$is\text{-}variable\text{-}on\text{-}domain(l) \rightarrow l;$

$\exists p \exists q(is\text{-}name\text{-}variable\text{-}on\text{-}domain(p) \wedge is\text{-}domain\text{-}of\text{-}parameter(q)$

$\wedge l = p \bullet \mathbf{ON} \bullet q)$

$\rightarrow$ **LIST-VAR** $(p, q);$

$\exists p \exists q(is\text{-}iterated\text{-}variable\text{-}on\text{-}domain(p) \wedge is\text{-}domain\text{-}of\text{-}parameter(q)$

$\wedge l = p \bullet \mathbf{ON} \bullet q)$

$\rightarrow$ **LIST-VAR** $(p, q))$

Function **LIST-VAR** (p, q) creates auxiliary relation $Temp(Cond, D, Iname, I)$, which tupies set the *domain of parameter* q and generate the list of variables with name p and indices from relation $Temp$.

LIST-VAR $(p, q) \sim$

$($ $is\text{-}name\text{-}of\text{-}unconditional\text{-}domain(q)$

$\rightarrow Temp := \mathbf{F}(Domains, D = q); Temp(I);$

$(\forall c(c \in Temp) \rightarrow q \bullet ($ $ReplaceAll(\circ,,,c) \bullet));$

$\exists x \exists y \exists a(is\text{-}name\text{-}of\text{-}unconditional\text{-}domain(x) \wedge is\text{-}name\text{-}of\text{-}index(y)$

$\wedge is\text{-}$ int- constant$(a) \wedge q = x \bullet / \bullet y \bullet = \bullet a)$

$\rightarrow Temp := \mathbf{F}(Domains, D = q \wedge Iname = y \wedge I = a); Temp(I);$

$(\forall c(c \in Temp) \rightarrow q \bullet ($ $ReplaceAll(\circ,,,c) \bullet));$

$\exists x \exists z(is\text{-}name\text{-}of\text{-}unconditional\text{-}domain(x) \wedge is\text{-}$ **list** $\text{-}index\text{-}expression(z)$

$\wedge q = x \bullet / (\bullet z \bullet))$

$\rightarrow (\forall y(y \in \overline{is}\text{-}name\text{-}of\text{-}index(z) \wedge \exists a(is\text{-}$ int- constant$(a) \wedge y \bullet = \bullet a \propto z))$

$\rightarrow Temp := \mathbf{F}(Domains, D = q \wedge Iname = y \wedge I = a); Temp(I);$

$(\forall c(c \in Temp) \rightarrow q \bullet ($ $ReplaceAll(\circ,,,c) \bullet)).)$

Function **GET-DEPENDENCE**(l), l - **list**-*initial-parameter* or *arithmetical expression* creates the list of arguments Arg for function of AM-machine $Def(Arg)$ where all the variables set in l are included. Thus in creation of Def attribute's value from relation $Relations$ the condition of the definite determination of the computation's arguments is considered and logical function $Def(Arg)$) is added. Function $Def(Arg)$ will be true $(\mathbf{T})$ if all $X_i \in Arg$ have already defined (i.e. their value is

the value of attribute $Value \neq \Lambda$) in other case it will be false ($\mathbb{F}$). Function $Def(Arg)$ is a built-in function of AM- machine and is performed in the process of its work.

$$\textbf{GET-DEPENDENCE}(l) \sim$$
$$(is\text{-}\mathrm{arithm}\text{-}\mathrm{expression}(l)$$
$$\rightarrow (\forall r(r \in \overline{is}\text{-}variable\text{-}on\text{-}domain(l) \vee r \in \overline{is}\text{-}name\text{-}of\text{-}scalar(l)) \rightarrow r);$$
$$\exists p \exists q(is\text{-}name\text{-}variable\text{-}on\text{-}domain(p) \wedge is\text{-}domain\text{-}of\text{-}parameter(q)$$
$$\wedge\, p \bullet \mathbf{ON} \bullet q \propto l$$
$$\rightarrow \textbf{LIST-VAR}\ (p, q);$$
$$\exists p \exists q(is\text{-}iterated\text{-}variable\text{-}on\text{-}domain(p) \wedge is\text{-}domain\text{-}of\text{-}parameter(q)$$
$$\wedge\, p \bullet \mathbf{ON} \bullet q \propto l$$
$$\rightarrow \textbf{LIST-VAR}\ (p, q))$$

2.8 The rules of AM-machine work

Relation $ARelations(Name, ItVal, Def, Cond, Var, Value, Func)$, $ARelations \subseteq Relations$ will be named *active* relation of AM-machine.

Transfer of the tuples from relation $Relations$ to $ARelations$ is carried out by built-in functions of AM-machine

CALL-PART$((p(actual\text{-}parameters))$,$p$- name of part and

NEXT-ITERATION $(n \bullet i)$, n - name of iteration index, i - value of iteration index.

Function **CALL-PART**$((p(actual\text{-}parameters))$ invokes the tuples of relation $Relations$ corresponding to computations of part p.

Function **NEXT-ITERATION** $(n \bullet i)$ invokes the tuples of relation $Relations$ corresponding to the next iteration step i+1 in iterative computation by iteration index n.

Definition of these functions are given below.

Interpretation of abstract program $A(P)$ by abstract machine AM consists of the initialization and stages of interpretation.

1. Initialization.

1) Function **CHECK-REASSIGNMENT**($Relations$) is carried out. It checks the condition of single-assigning the value to the variable of program P.

2) Function **CALL-PART**($main$) is done. $Main$ - name of entry node of the calls' graph $G(V,E)$.

2. Stages of interpretation.

1) Set $ArgDefined$ is chosen from relation $Relation$. $ArgDefined$ is a set of tuples which have all the arguments computed but the values of the variables haven't computed yet. The tuples from $ArgDefined$ which computation depends on the condition $Cond$ and $Cond$=$\mathbb{F}$ are eliminated from $ArgDefined$ and $ARelations$.

$$ArgDefined := \mathbf{F}(ARelations, Def = \mathbb{T} \wedge Value = \Lambda);$$
$$ArgDefined := ArgDefined - \mathbf{F}(ArgDefined, Cond = \mathbb{F});$$
$$ARelations := ARelations - \mathbf{F}(ARelations, Cond = \mathbb{F})$$

2) Arbitrary subset $NonDeterm \subseteq ArgDefined$ is chosen. Arbitrariness of such a choice is determined by non-determination of AM-machine and allows to specify not the only process of program P fulfillment but the class of all possible computations.

$$NonDeterm \subseteq ArgDefined$$

3) Function *func* is computed for every tuple $\langle name \circ itval \circ def \circ cond \circ var \circ value \circ func \rangle$ from *NonDeterm*. This function is set in the field of attribute *Func* of the same tuple. Besides some variants are possible:

a) *func* – call of simple part *name*(*actual-parameters*). The following function of *AM*-machine is carried out in this case:

CALL-PART(*name*(*actual-parameters*))

b) *func* – call of external function *name*(*actual-parameters*). The following function of *AM*-machine is carried out in this case:

CALL-PART(*name*(*actual-parameters* **RESULT** *name*))

c) *func* – arithmetical expression, call of reduction function or operator **INPUT**.

Arithmetical expression and reduction function are computed (semantics of reduction functions coincides with common mathematical definition of these functions), operator **INPUT** is carried out and obtained value *val* substitutes value $value = \Lambda$ of considered tuple. Besides if there is tuple $\langle cond \circ var \circ value_1 \circ file \rangle$ in relation *Output* then the value $value_1 = \Lambda$ of this tuple is substituted to *val*.

Formal definition of computation from *NonDeterm*:

COMPUTE(*NonDeterm*);

COMPUTE(*nondeterm*) ~

($nondeterm = \langle name \circ itval \circ def \circ cond \circ var \circ value \circ func \rangle$

$\rightarrow$ (*is - call - of - part*(*func*)

$\rightarrow$ **CALL-PART**(*func*); *Delete*(*nondeterm*, *NonDeterm*);

$\exists f \exists l$(*is - name - external - function*(f)

$\wedge$ *is* - list - *initial - parameter*(l) $\wedge$ $f \bullet (\bullet l \bullet) = func$)

$\rightarrow$ **CALL-PART**($f \bullet (\bullet l \bullet$ **RESULTS** $\bullet l \bullet)$); *Delete*(*nondeterm*, *NonDeterm*);

is - arithm - expression(*func*)

$\rightarrow$ *value* := **EVALUATE-EXPR**(*func*); **UPDATE**(*Output*, *value*);

is - call - to - reduction - function(*func*)

$\rightarrow$ *value* := **REALIZE**(*func*); **UPDATE**(*Output*, *value*);

func = **INPUT**

$\rightarrow$ *value* := **INPUT**(*Input*, *Var*); **UPDATE**(*Output*, *value*)))

Specification of function **UPDATE**(*Output*, *value*) and **INPUT**(*Input*,*var*) is evident enough and accurate definition of function **REALIZE**(*func*) is determined by realization of *AM*-machine.

4) Set *OutDefined* is chosen from relation *Output*. *OutDefined* is a set of tuples which have all the arguments computed. The tuples from *Out Defined* which computation depends on the condition *Cond* and *Cond*=**F** are eliminated from *OutDefined*.

OutDefined := **F**(*Output*, *Value* $\neq \Lambda$);

$OutDefined := OutDefined - \mathbf{F}(Output, Cond = \mathbf{F})$;

5) Arbitrary subset $NonDetermOut \subseteq OutDefined$ is chosen. The value *value* is output into file *file* for every tuple $\langle cond \circ x \circ i \circ value \circ file \rangle$ from *NonDetermOut*.

$NonDetermOut \subseteq OutDefined$
$OutDefined := OutDefined - NonDetermOut$
OUTPUT(*NonDetermOut*)

6) Iterative computations set by the tuples of relation *ARelations* for the next values of iteration indices (if the iterative computations aren't finished in accordance with the condition of an exit from the iteration) are invoked.

$ActivIter := \mathbf{F}(ARelations,\ ItVal \neq \varnothing\varnothing)$;
$ActivIterIndex :=$ IT-INDEX(*ActivIter*(*ItVal*));
NEXT-ITERATION(*ActivIterIndex*)

7) Condition **CHECK-STOP** of *AM*-machine stop is checked.

If all the necessary computations are finished and the values of all the variables which are to be computed have been computed then *AM*-machine finishes working it NORMA'lly and passes over to state *stop*.

If there are variables which values are to be computed but new computations aren't possible then *AM*-machine finishes working it AbNORMA'lly and passes over to state *error*.

In other case the next stage of iteration is carried out.

CHECK-STOP ~
$ActivIter := \mathbf{F}(ARelations, ItVal \neq \varnothing\varnothing)$;
$($ $\mathbf{F}(ActivIter, Var = \varnothing)(Cond) = \mathbf{T} \wedge ARelations(Value) \neq \Lambda$
$\rightarrow stop$;

$\mathbf{F}(ActivIter, Var = \varnothing)(Cond) = \mathbf{T} \wedge \exists\, \alpha(\alpha \in ARelations(Value) \wedge \alpha \neq \Lambda$
$\rightarrow error$;
$\rightarrow nextAMstep)$

□ *Checking the condition of single-assignment* **CHECK-REASSIGNMENT**(*Relations*).

If there exist even two coincided variables which values are to be computed and the conditions of theses variables' computation don't come into the conflict then the condition of single-assignment is broken.

CHECK-REASSIGNMENT(*Relations*) ~
$(\exists\, \alpha_1 \exists\, \alpha_2 (\alpha_1 . \alpha_2 \in Relations$
$\wedge\, \alpha_1 = \langle name_1 \circ itval_1 \circ def_1 \circ cond_1 \circ \mathrm{var}_1 \circ value_1 \circ func_1 \rangle$
$\wedge\, \alpha_2 = \langle name_2 \circ itval_2 \circ def_2 \circ cond_2 \circ \mathrm{var}_2 \circ value_2 \circ func_2 \rangle$
$\wedge\, \mathrm{var}_1 \cap \mathrm{var}_2 \neq \varnothing$
$\wedge\, (cond_1 \wedge \neg(cond_2) \equiv \mathbf{F})$
$\wedge\, itval_1 = itval_2$
$\rightarrow \mathbf{F})$

□ *Invoking the tuples while calling the part* **CALL-PART**(c).

Correspondence of actual and formal parameters of c part's call is set and the actual parameters are passed to the tuples of the part. The tuples setting iterative computations are initialized and all the tuples are transfered into relation *ARelations* (with the renaming of the local variables).

CALL-PART(c) ~

$(\exists\, p \exists\, q(is\text{-}name\text{-}of\text{-}part(p) \wedge is\text{-}actual\text{-}parameters(p) \wedge p \bullet (\bullet q \bullet) = c)$

$\rightarrow$ *RPart*:= **F**(*Relations*, *Name* = p);

f:=**GET-FORMAL**(p);

RActul:= **ACTUAL-FORMAL**(*RPart*,f,p,q);

RReady:=**INI-ITERATION**(*RAll*);

RENAME-IN-CALL(*RReady*) **ADD-TO** *ARelations*)

Function **GET-FORMAL**(p) analyses the graph of calls $G(V,E)$ and give the formal parameters of part p.

Function **ACTUAL-FORMAL**(*RPart*,f,p,q) carries out passing actual parameters f to the tuples of relation *Rpart* from the part with name p and formal parameters q.

ACTUAL-FORMAL(*RPart*,f,p,q); ~

CHECK-FORMAL-ACTUAL(f,q);

$(\exists\, g \exists\, h \exists\, s(is\text{-}\text{list}\text{-}name(g) \wedge is\text{-}\text{list}\text{-}name(h)$

$\wedge g \bullet \mathbf{RESULTS} \bullet h = f$

$\wedge is\text{-}\text{list}\text{-}initial\text{-}parameter(r) \wedge is\text{-}\text{list}\text{-}parameter\text{-}result(s)$

$\wedge r \bullet \mathbf{RESULTS} \bullet s = q$

$\rightarrow$ *Rpart* := **MAKE-IN**(p,*RPart*,g,r); *Rpart* := **MAKE-OUT**(p,*RPart*,h,s))

Function **MAKE-IN**(p,*RPart*,g,r) chooses the next elements from the lists of formal initial parameters g and actual initial parameters r.

MAKE-IN(p,*RPart*,g,r) ~

(*is*-name(g) $\rightarrow$ **MAKE-IN-PARAMETER** (p,*RPart*,g,r,);

is-list-name(g) $\rightarrow$ **MAKE-IN-PARAMETER**(p,*RPart*,*head*(g),*head*(r));

MAKE-IN(p,*RPart*,*tail*(g),*tail*(r));)

Function **MAKE-OUT**(p,*RPart*,g,r) chooses the next elements from the lists of formal parameters results g and actual parameters results r.

MAKE-OUT(p,*RPart*,g,r) ~

(*is*-name(g) $\rightarrow$ **MAKE-OUT-PARAMETER** (p,*RPart*,g,r,);

is-list-name(g) $\rightarrow$ **MAKE-OUT-PARAMETER** (p,*RPart*,*head*(g),*head*(r));

MAKE-OUT (p,*RPart*,*tail*(g),*tail*(r));)

Function **MAKE-IN-PARAMETER**(p,*RPart*,g,r,) modifies the tuples of relation *Rpart* from the part with name p while substituting initial formal parameter g to initial actual parameter r.

MAKE-IN-PARAMETER$(p,RPart,g,r,) \sim$

$(is\text{-}\mathrm{arithm}\text{-}expression(r)$

$\rightarrow val =$ **EVALUATE-EXPR**$(r)\, p \circ \mathbf{T} \circ \mathbf{T} \circ g \circ val \circ val$ **ADD-TO** $RPart;$

$is\text{-}name\text{-}of\text{-}external\text{-}simple\text{-}part(r) \rightarrow ReplaceAll(g,r,RPart);$

$is\text{-}name\text{-}of\text{-}external\text{-}function(r) \rightarrow ReplaceAll(g,r,RPart);$

$\exists n \exists q(is\text{-}name\text{-}of\text{-}\mathrm{variable}\text{-}on\text{-}domain(n) \wedge is\text{-}domian\text{-}of\text{-}parameter(q)$

$\wedge r = n \bullet \mathbf{ON} \bullet q)$

$\rightarrow (\exists x \exists y(is\text{-}name\text{-}of\text{-}unconditional\text{-}domain(x) \wedge is\text{-}\mathrm{list}\text{-}ind\text{-}expression(y)$

$\wedge q = x \bullet / \bullet y \bullet))$

$\rightarrow name_1 := uname(P); Replace(q,name_1,r); new\text{-}domain(name_1,x,y);$

$name_2 := uname(P);$

$Include(\mathbf{VARIABLE} \bullet name_2 \bullet \mathbf{DEFINED\ ON} \bullet name_1 \bullet .,p);$

VAR-DECLARATION$(p);$

$Include(\mathbf{FOR} \bullet name_1 \bullet \mathbf{ASSUME} \bullet name_2 \bullet = n \bullet [\bullet y \bullet].,p);$

$Include(\mathbf{FOR} \bullet name_1 \bullet \mathbf{ASSUME} \bullet g \bullet = name_2 \bullet .,p);$

REBUILD-INDEX(p); **OPERATORS-TO-RELATIONS**$(p);$

$is\text{-}name\text{-}of\text{-}unconditional\text{-}domain(q)$

$\rightarrow name_2 := uname(P);$

$Include(\mathbf{VARIABLE} \bullet name_2 \bullet \mathbf{DEFINED\ ON} \bullet q \bullet .,p);$

VAR-DECLARATION$(p);$

$Include(\mathbf{FOR} \bullet q \bullet \mathbf{ASSUME} \bullet name_2 \bullet = n \bullet .,p);$

$Include(\mathbf{FOR} \bullet q \bullet \mathbf{ASSUME} \bullet g \bullet = name_2 \bullet .,p);$

REBUILD-INDEX(p); **OPERATORS-TO-RELATIONS**$(p))$)

Function **NEW-DOMAIN**(n,x,y) builds new domain with name n from the domain with name x and list of modifications y. This function is an analogous of the superposition of functions **RECT-N-DOMAIN-TRANSFORM**(o) and **RECT-N-DOMAIN-EVALUATE**(o) defined above.

Function **MAKE-OUT-PARAMETER**$(p,RPart,g,r)$ modifies the tuples of relation $Rpart$ from the part with name p while substituting formal parameter-result g to actual parameter-result r.

MAKE-OUT-PARAMETER $(p,RPart,g,r,) \sim$

$(is\text{-}name\text{-}of\text{-}scalar(r) \rightarrow p \circ \mathbf{T} \circ \mathbf{T} \circ g \circ val \circ val$ **ADD-TO** $RPart;$

$\exists n \exists q(is\text{-}name\text{-}of\text{-}\mathrm{variable}\text{-}on\text{-}domain(n) \wedge is\text{-}domian\text{-}of\text{-}parameter(q)$

$\wedge r = n \bullet \mathbf{ON} \bullet q)$

$\rightarrow (\exists x \exists y(is\text{-}name\text{-}of\text{-}unconditional\text{-}domain(x) \wedge is\text{-}\mathrm{list}\text{-}ind\text{-}expression(y)$

$\wedge q = x \bullet / \bullet y \bullet))$

$\rightarrow name_1 := uname(P); Replace(q,name_1,r); new\text{-}domain(name_1,x,y);$

$name_2 := uname(P);$

$Include(\mathbf{VARIABLE} \bullet name_2 \bullet \mathbf{DEFINED\ ON} \bullet name_1 \bullet .,p);$

VAR-DECLARATION$(p);$

$Include(\mathbf{FOR} \bullet name_1 \bullet \mathbf{ASSUME} \bullet n \bullet = name_2 \bullet .,p);$

$Include(\mathbf{FOR} \bullet name_1 \bullet \mathbf{ASSUME} \bullet name_2 \bullet = \bullet g \bullet ., p);$

REBUILD-INDEX(p); **OPERATORS-TO-RELATIONS**(p);

$is\text{-}name\text{-}\underline{of}\text{-}unconditional\text{-}domain(q)$

$\rightarrow name_2 := uname(P);$

$Include(\mathbf{VARIABLE} \bullet name_2 \bullet \mathbf{DEFINED\ ON} \bullet q \bullet ., p);$

VAR-DECLARATION(p);

$Include(\mathbf{FOR} \bullet q \bullet \mathbf{ASSUME} \bullet n \bullet = name_2 \bullet ., p);$

$Include(\mathbf{FOR} \bullet q \bullet \mathbf{ASSUME} \bullet name_2 \bullet = g \bullet ., p);$

REBUILD-INDEX(p); **OPERATORS-TO-RELATIONS**(p)))

□ *Invoking tuples at the next iteration step* **NEXT-ITERATION**(n).

If the next iteration step on iteration index n is terminated and the exit condition of iteration hasn't been performed then the tuples are invoked for the next iteration step.

NEXT-ITERATION$(n) \sim$

$OneIter := \mathbf{F}(ARelations, ItVal = n);$

$(\mathbf{F}(OneIter, Var = \varnothing)(Cond) = \mathbf{F} \wedge \mathbf{F}(OneIter, Var \neq \varnothing)(Value) \neq \Lambda$

$\rightarrow$ **GET-NEXT-ITERATION**(n);

$\rightarrow$ **RESULT-ITERATION**$(n, max(OneIter(ItVal))))$

The tuples specifying the *body of iteration* are chosen from relation $Relations$.

GET-NEXT-ITERATION$(n) \sim$

$NextIter := \mathbf{F}(Relations, ItVal = n \bullet 1);$

$(\forall c(c \in NextIter \wedge c = \langle name \circ itval \circ def \circ cond \circ \mathrm{var} \circ value \circ func \rangle$

$\rightarrow$ **UPDATE**$(c, \langle name \circ n \circ i+1 \circ def \circ cond \circ \mathrm{var} \circ value \circ func \rangle, NextIter);$

INI-ITERATION$(NextIter)$ **ADD-TO** $ARelations)$

Function **INI-ITERATION**(R) initializes iterative computations set by the tuples of relation R

INI-ITERATION$(R) \sim$

$(\forall c(c \in R \wedge c = \langle name \circ itval \circ def \circ cond \circ \mathrm{var} \circ value \circ func \rangle \wedge itval = \varnothing\ \varnothing$

$\rightarrow;$

$(\forall c(c \in RAll \wedge c = \langle name \circ itval \circ def \circ cond \circ \mathrm{var} \circ value \circ func \rangle \wedge itval = i \bullet a$

$\rightarrow ReplaceAll(i, a, c))$

Function **RESULT-ITERATION**$(n,a))$ substitutes the name of iteration index n to its maximum value a in all the tuples of relation $ARelation$ which don't set iterative computations. Passing the results of iterative computations is done in this way.

There is auxiliary function **IT-INDEX** $(n \bullet i) \sim n$.

2.9 The example of AM-machine work.

Consider program P where the principal constructions of the NORMA language are given. Note that this program is demonstrative and computational formulae used in the program don't essentially describe any known numerical method.

```
MAIN PART One.
BEGIN
Oji:( Ojj; Oii ).
Oij:( Oi:(i=1..n-1); Oj:(j=1..n-1) ).
Ojj:Oj/j=1..n-2. Oii:Oi/Oi-RIGHT(n-3).
        INDEX i,j.
        DOMAIN PARAMETERS n = 4.
VARIABLE a,T,Ts,Tn DEFINED ON Oij. VARIABLE x,y DEFINED ON Oi.
VARIABLE Cond DEFINED ON Oji. VARIABLE b DEFINED ON Oj.
OjiYES , OjiNOT : Oji/ Ts < 0.5.
        INPUT  a(FILE='data.dat') ON Oij, b(FILE='data.dat') ON Oj.
        INPUT  T ON Oij/i=1,j=2..n-1,  T ON Oij/j=1,i=2..n-1.
        OUTPUT T(FILE='results') ON Oij/i=n-1,j=n-1.
        OUTPUT x(FILE='results') ON Oi.
FOR Oi  ASSUME  x = y + SUM((Oj)a*b); y = b[j=i].
FOR Oij/i=2..n-1,j=2..n-1 ASSUME  T = T[i-1]+T[j-1]; Ts = SIN(T).
FOR Oij/i=1,j=1..n-1 ASSUME Ts = SIN(i+j).
FOR Oij/j=1,i=2..n-1 ASSUME Ts = SIN(i-j).
FOR Oi  ASSUME  COMPUTE Two(Ts ON Oij/j=i RESULT Tn ON Oij/i=i).
        OUTPUT Tn(FILE='results') ON Oij, Cond ON OjiYES.
FOR OjiYES ASSUME  Cond = Tn.
FOR OjiNOT ASSUME  Cond = Ts.
END PART.
PART Two. Ts RESULT Tn
BEGIN
Oi:(i=1..M).   DOMAIN PARAMETERS M = 3.
VARIABLE Ts,Tn,T DEFINED ON Oi. VARIABLE Eps.
INPUT Eps. INPUT T(FILE='data.dat') ON Oi.
ITERATION Tn ON N.
        INITIAL N = 0 :
        FOR Oi ASSUME Tn = Ts.
        END INITIAL
FOR Oi ASSUME Tn = Tn[N-1]/M.
EXIT WHEN  ABS(Tn[i=3,N]) < Eps.
END ITERATION N.
END PART.
```

Input file data.dat:

```
b(j=1..2)          =  5.0, 7.0;
a(i=1..3,j=1..3)   =  2.0, 3.0, -4.0,
                      3.0, 4.0, -1.0,
                      3(5.0);
T(i=1..3)          =  3(-1.1);
b(j=3)             =  9.0;
```

Input file norma.dat:

```
Eps                =  1.0E-6;
T(i=2..3,j=1)      =  3.3, 4.4;
T(i=1,j=2..3)      =  1.1, 2.2;
```

The result of performing function LOCAL-VARIABLES-RENAME(o),o=One,Two - the program and input files get the form (we consider function $uname(P)$ engenders unique names by adding zeroes as the last identificator's symbols):

```
MAIN PART One.
```

```
BEGIN
Oji0:( Ojj0; Oii0 ).
Oij0:( Oi0:(i0=1..n0-1); Oj0:(j0=1..n0-1) ).
Ojj0:Oj0/j0=1..n0-2. Oii0:Oi0/Oi0-RIGHT(n0-3).
        INDEX i0,j0.
        DOMAIN PARAMETERS n0 = 4.
VARIABLE a0,T0,Ts0,Tn0 DEFINED ON Oij0. VARIABLE x0,y0 DEFINED ON Oi0.
VARIABLE Cond0 DEFINED ON Oji0. VARIABLE b0 DEFINED ON Oj0.
OjiYES0 , OjiNOT0 : Oji0/ Ts0 < 0.5.
        INPUT  a0(FILE='data.dat') ON Oij0, b0(FILE='data.dat') ON Oj0.
        INPUT  T0 ON Oij0/i0=1,j0=2..n0-1,  T0 ON Oij0/j0=1,i0=2..n0-1.
        OUTPUT T0(FILE='results') ON Oij0/i0=n0-1,j0=n0-1.
        OUTPUT x0(FILE='results') ON Oi0.
FOR Oi0  ASSUME  x0 = y0 + SUM((Oj0)a0*b0); y0 = b0[j0=i0].
FOR Oij0/i0=2..n0-1,j0=2..n0-1 ASSUME  T0 = T0[i0-1]+T[j0-1]; Ts0 = SIN(T0).
FOR Oij0/i0=1,j0=1..n0-1 ASSUME Ts0 = SIN(i0+j0).
FOR Oij0/j0=1,i0=2..n0-1 ASSUME Ts0 = SIN(i0-j0).
FOR Oi0  ASSUME
        COMPUTE Two(Ts0 ON Oij0/j0=i0 RESULT Tn0 ON Oij0/i0=i0).
        OUTPUT Tn0(FILE='results') ON Oij0, Cond0 ON OjiYES0.
FOR OjiYES0 ASSUME  Cond0 = Tn0.
FOR OjiNOT0 ASSUME  Cond0 = Ts0.
END PART.
PART Two. Ts00 RESULT Tn00
BEGIN
Oi00:(i00=1..M00).   DOMAIN PARAMETERS M00 = 3.
VARIABLE Ts00,Tn00,T00 DEFINED ON Oi00. VARIABLE Eps00.
INPUT Eps00. INPUT T00(FILE='data.dat') ON Oi00.
ITERATION Tn00 ON N00.
        INITIAL N00 = 0 :
        FOR Oi00 ASSUME Tn00 = Ts00.
        END INITIAL
FOR Oi00 ASSUME Tn00 = Tn00[N00-1]/M00.
EXIT WHEN  ABS(Tn00[i00=3,N00]) < Eps00.
END ITERATION N00.
END PART.
```

Input file data.dat:

```
b0(j0=1..2)          =  5.0, 7.0;
a0(i0=1..3,j0=1..3)  =  2.0, 3.0, -4.0,
                        3.0, 4.0, -1.0,
                        3(5.0);
T00(i00=1..3)        =  3(-1.1);
b0(j0=3)             =  9.0;
```

Input file norma.dat:

```
Eps                  =  1.0E-6;
T0(i0=2..3,j0=1)     =  3.3, 4.4;
T0(i0=1,j0=2..3)     =  1.1, 2.2;
```

The result of caryying out function DEF-INDEX-SPACE(o), o =One, Two— constructing

$$IndSpace = \{IndSpace_i\}, \quad i = 1,\ldots,2;$$
$$IndSpace_1 = \mathbf{One} \bullet i0 \bullet j0$$
$$IndSpace_2 = \mathbf{Two} \bullet i00$$

and deleting declaration **INDEX i0,j0** from the text of the program.

The result of performing function PAR-DOMAIN-EXCLUDE(o),o=One,Two:

```
MAIN PART One.
BEGIN
```

```
Oji0:( Ojj0; Oii0 ).
Oij0:( Oi0:(i0=1..4-1); Oj0:(j0=1..4-1) ).
Ojj0:Oj0/j0=1..4-2. Oii0:Oi0/Oi0-RIGHT(4-3).
VARIABLE a0,T0,Ts0,Tn0 DEFINED ON Oij0. VARIABLE x0,y0 DEFINED ON Oi0.
VARIABLE Cond0 DEFINED ON Oji0. VARIABLE b0 DEFINED ON Oj0.
OjiYES0 , OjiNOT0 : Oji0/ Ts0 < 0.5.
        INPUT  a0(FILE='data.dat') ON Oij0, b0(FILE='data.dat') ON Oj0.
        INPUT  T0 ON Oij0/i0=1,j0=2..4-1,  T0 ON Oij0/j0=1,i0=2..4-1.
        OUTPUT T0(FILE='results') ON Oij0/i0=4-1,j0=4-1.
        OUTPUT x0(FILE='results') ON Oi0.
FOR Oi0  ASSUME  x0 = y0 + SUM((Oj0)a0*b0); y0 = b0[j0=i0].
FOR Oij0/i0=2..4-1,j0=2..4-1 ASSUME  T0 = T0[i0-1]+T[j0-1]; Ts0 = SIN(T0).
FOR Oij0/i0=1,j0=1..4-1 ASSUME Ts0 = SIN(i0+j0).
FOR Oij0/j0=1,i0=2..4-1 ASSUME Ts0 = SIN(i0-j0).
FOR Oi0  ASSUME
        COMPUTE Two(Ts0 ON Oij0/j0=i0 RESULT Tn0 ON Oij0/i0=i0).
        OUTPUT Tn0(FILE='results') ON Oij0, Cond0 ON OjiYES0.
FOR OjiYES0 ASSUME  Cond0 = Tn0.
FOR OjiNOT0 ASSUME  Cond0 = Ts0.
END PART.
PART Two. Ts00 RESULT Tn00
BEGIN
Oi00:(i00=1..3).
VARIABLE Ts00,Tn00,T00 DEFINED ON Oi00. VARIABLE Eps00.
INPUT Eps00. INPUT T00(FILE='data.dat') ON Oi00.
ITERATION Tn00 ON N00.
        INITIAL N00 = 0 :
        FOR Oi00 ASSUME Tn00 = Ts00.
        END INITIAL
FOR Oi00 ASSUME Tn00 = Tn00[N00-1]/3.
EXIT WHEN  ABS(Tn00[i00=3,N00]) < Eps00.
END ITERATION N00.
END PART.
```

The result of performing function CONST-EXPR-EVALUATE(o),o=One,Two (part Two isn't changed):

```
MAIN PART One.
BEGIN
Onew0:((i0=1);(j0=2..3)). Onew1:((j0=1);(i0=2..3)). Onew2:((i0=3);(j0=3)).
Onew3:((i0=2..3);(j0=2..3)). Onew4:((i0=1);(j0=1..3)). Onew5:((j0=1);(i0=2..3)).
Oji0:( (j0=1..2); i0=1..2 ).
Oij0:(    (i0=1..3);    (j0=1..3) ).
        Oi0:(i0=1..3). Oj0:(j0=1..3).
Ojj0:(j0=1..2). Oii0:(i0=1..2).
VARIABLE a0,T0,Ts0,Tn0 DEFINED ON Oij0. VARIABLE x0,y0 DEFINED ON Oi0.
VARIABLE Cond0 DEFINED ON Oji0. VARIABLE b0 DEFINED ON Oj0.
OjiYES0 , OjiNOT0 : Oji0/ Ts0 < 0.5.
        INPUT  a0(FILE='data.dat') ON Oij0, b0(FILE='data.dat') ON Oj0.
        INPUT  T0 ON Onew0,  T0 ON Onew1.
        OUTPUT T0(FILE='results') ON Onew2.
        OUTPUT x0(FILE='results') ON Oi0.
FOR Oi0  ASSUME  x0 = y0 + SUM((Oj0)a0*b0). FOR Oi0  ASSUME y0 = b0[j0=i0].
FOR Onew3 ASSUME T0 = T0[i0-1]+T[j0-1]. FOR Onew3 ASSUME  Ts0 = SIN(T0).
FOR Onew4 ASSUME Ts0 = SIN(i0+j0).    FOR Onew5 ASSUME Ts0 = SIN(i0-j0).
FOR Oi0  ASSUME
        COMPUTE Two(Ts0 ON Oij0/j0=i0 RESULT Tn0 ON Oij0/i0=i0).
        OUTPUT Tn0(FILE='results') ON Oij0, Cond0 ON OjiYES0.
FOR OjiYES0 ASSUME  Cond0 = Tn0.
```

FOR OjiNOT0 ASSUME Cond0 = Ts0.
END PART.

The result of perfotming functions **RECT-DOMAIN-EVALUATE**(o), **DIAG-DOMAIN-EVALUATE**(o),o=**One,Two** is filling in relation *Domains*:

Domains			
Cond	*D*	*Iname*	*I*
T	Onew0	i0 j0	1 2
T	Onew0	i0 j0	1 3
T	Onew1	i0 j0	2 1
T	Onew1	i0 j0	3 1
T	Onew2	i0 j0	3 3
T	Onew3	i0 j0	2 2
T	Onew3	i0 j0	2 3
T	Onew3	i0 j0	3 2
T	Onew3	i0 j0	3 3
T	Onew4	i0 j0	1 1
T	Onew4	i0 j0	1 2
T	Onew4	i0 j0	1 2
T	Onew5	i0 j0	2 1
T	Onew5	i0 j0	3 1
T	Oji0	i0 j0	1 1
T	Oji0	i0 j0	1 2
T	Oji0	i0 j0	2 1
T	Oji0	i0 j0	2 2
T	Oij0	i0 j0	1 1

Cond	*D*	*Iname*	*I*
T	Oij0	i0 j0	1 2
T	Oij0	i0 j0	1 3
T	Oij0	i0 j0	2 1
T	Oij0	i0 j0	2 2
T	Oij0	i0 j0	2 3
T	Oij0	i0 j0	3 1
T	Oij0	i0 j0	3 2
T	Oij0	i0 j0	3 3
T	Oi0	i0	1
T	Oi0	i0	2
T	Oi0	i0	3
T	Oj0	j0	1
T	Oj0	j0	2
T	Oj0	j0	3
T	Oii0	i0	1
T	Oii0	i0	2
T	Ojj0	j0	1
T	Ojj0	j0	2
T	Oi00	i00	1
T	Oi00	i00	2
T	Oi00	i00	3

Besides declaration of the following domains are deleted from the text of part **One**:

Onew0:((i0=1);(j0=2..3)). Onew1:((j0=1);(i0=2..3)). Onew2:((i0=3);(j0=3)).
Onew3:((i0=2..3);(j0=2..3)). Onew4:((i0=1);(j0=1..3)). Onew5:((j0=1);(i0=1..3)).
Oji0:((j0=1..2); i0=1..2).
Oij0:((i0=1..3); (j0=1..3)).
Oi0:(i0=1..3). Oj0:(j0=1..3).
Ojj0:(j0=1..2). Oii0:(i0=1..2).

and the declaration of the domain given below is deleted from the text of part **Two**:
Oi00:(i00=1..3).

The result of performing functions **VAR-DECLARATION**(o) and **REBUILD-INDEX**(o),o=**One,Two** - modification of program and filling in relation *Variables*:

MAIN PART One.
BEGIN
OjiYES0 , OjiNOT0 : Oji0/ Ts0(i0,j0) < 0.5.
INPUT a0(i0,j0)(FILE='data.dat') ON Oij0, b0(j0)(FILE='data.dat') ON Oj0.
INPUT T0(i0,j0) ON Onew0, T0(i0,j0) ON Onew1.
OUTPUT T0(i0,j0)(FILE='results') ON Onew2.
OUTPUT x0(i0)(FILE='results') ON Oi0.
FOR Oi0 ASSUME x0(i0) = y0(i0) + SUM((Oj0)a0(i0,j0)*b0(j0)).
FOR Oi0 ASSUME y0(i0) = b0(i0).
FOR Onew3 ASSUME T0(i0,j0) = T0(i0-1,j0)+T(i0,j0-1).
FOR Onew3 ASSUME Ts0(i0,j0) = SIN(T0(i0,j0)).

```
FOR Onew4 ASSUME Ts0(i0,j0) = SIN(i0+j0).
FOR Onew5 ASSUME Ts0(i0,j0) = SIN(i0-j0).
FOR Oi0  ASSUME
        COMPUTE Two(Ts0 ON Oij0/j0=i0 RESULT Tn0 ON Oij0/i0=i0).
        OUTPUT Tn0(i0,j0)(FILE='results') ON Oij0, Cond0(i0,j0) ON OjiYES0.
FOR OjiYES0 ASSUME  Cond0(i0,j0) = Tn0(i0,j0).
FOR OjiNOT0 ASSUME  Cond0(i0,j0) = Ts0(i0,j0).
END PART.
PART Two. Ts00 RESULT Tn00
BEGIN
INPUT Eps00. INPUT T00(i00)(FILE='data.dat') ON Oi00.
ITERATION Tn00 ON N00.
        INITIAL N00 = 0 :
        FOR Oi00 ASSUME Tn00(i00,N00) = Ts00(i00).
        END INITIAL
FOR Oi00 ASSUME Tn00(i00,N00) = Tn00[i00,N00-1]/3.
EXIT WHEN  ABS(Tn00[3,N00]) < Eps00.
END ITERATION N00.
END PART.
```

Variables		
D	*X*	*Iter*
Oij0	a0	Λ
Oij0	T0	Λ
Oij0	Ts0	Λ
Oij0	Tn0	Λ
Oi0	x0	Λ
Oi0	y0	Λ
Oji0	Cond0	Λ
Oj0	b0	Λ
Oi00	Ts00	Λ
Oi00	Tn00	N00
Oi00	T00	Λ
∅	Eps0	Λ

The result of performing function COND-DOMAIN-EVALUATE(o),o=**One**,**Two** is adding tuples to relation *Domains*:

Domains			
Cond	*D*	*Iname*	*I*
(Ts0(1,1)<0.5)	OjiYES0	i0 j0	1 1
(Ts0(1,2)<0.5)	OjiYES0	i0 j0	1 2
(Ts0(2,1)<0.5)	OjiYES0	i0 j0	2 1
(Ts0(2,2)<0.5)	OjiYES0	i0 j0	2 2
¬(Ts0(1,1)<0.5)	OjiNOT0	i0 j0	1 1
¬(Ts0(1,2)<0.5)	OjiNOT0	i0 j0	1 2
¬(Ts0(2,1)<0.5)	OjiNOT0	i0 j0	2 1
¬(Ts0(2,2)<0.5)	OjiNOT0	i0 j0	2 2

and deleting declaration of the following domain from the text of part **One**
OjiYes0 , OjiNOT0 : Oji0/Ts0(i0,j0) < 0.5.

The result of performing function **INPUT-DECLARATION**(o), **OUTPUT-DECLARATION**(o),o=**One,Two** - modification of the program and filling in relations *Input*, *Output*, *Relations*.

```
MAIN PART One.
BEGIN
FOR Oi0  ASSUME  x0(i0) = y0(i0) + SUM((Oj0)a0(i0,j0)*b0(j0)).
FOR Oi0  ASSUME  y0(i0) = b0(i0).
FOR Onew3 ASSUME T0(i0,j0) = T0(i0-1,j0)+T(i0,j0-1).
FOR Onew3 ASSUME Ts0(i0,j0) = SIN(T0(i0,j0)).
FOR Onew4 ASSUME Ts0(i0,j0) = SIN(i0+j0).
FOR Onew5 ASSUME Ts0(i0,j0) = SIN(i0-j0).
FOR Oi0  ASSUME
   COMPUTE Two(Ts0 ON Oij0/j0=i0 RESULT Tn0 ON Oij0/i0=i0).
FOR OjiYES0 ASSUME  Cond0(i0,j0) = Tn0(i0,j0).
FOR OjiNOT0 ASSUME  Cond0(i0,j0) = Ts0(i0,j0).
END PART.
PART Two. Ts00 RESULT Tn00
BEGIN
ITERATION Tn00 ON N00.
 INITIAL N00 = 0 :
 FOR Oi00 ASSUME Tn00(i00,N00) = Ts00(i00).
 END INITIAL
FOR Oi00 ASSUME Tn00(i00,N00) = Tn00[i00,N00-1]/3.
EXIT WHEN  ABS(Tn00[3,N00]) < Eps00.
END ITERATION N00.
END PART.
```

Input		
Var	*Value*	*File*
a0(1,1)	Λ	'data.dat'
a0(1,2)	Λ	'data.dat'
a0(1,3)	Λ	'data.dat'
a0(2,1)	Λ	'data.dat'
a0(2,2)	Λ	'data.dat'
a0(2,3)	Λ	'data.dat'
a0(3,1)	Λ	'data.dat'
a0(3,2)	Λ	'data.dat'
a0(3,3)	Λ	'data.dat'
b0(1)	Λ	'data.dat'
b0(2)	Λ	'data.dat'
b0(3)	Λ	'data.dat'
T0(1,2)	Λ	'norma.dat'
T0(1,3)	Λ	'norma.dat'
T0(2,1)	Λ	'norma.dat'
T0(3,1)	Λ	'norma.dat'
Eps00	Λ	'norma.dat'
T00(1)	Λ	'data.dat'
T00(2)	Λ	'data.dat'
T00(3)	Λ	'data.dat'

Output			
Cond	*Var*	*Value*	*File*
T	T0(3,3)	Λ	'rezults'
T	x0(1)	Λ	'rezults'
T	x0(2)	Λ	'rezults'
T	x0(3)	Λ	'rezults'
T	Tn0(1,1)	Λ	'rezults'
T	Tn0(1,2)	Λ	'rezults'
T	Tn0(1,3)	Λ	'rezults'
T	Tn0(2,1)	Λ	'rezults'
T	Tn0(2,2)	Λ	'rezults'
T	Tn0(2,3)	Λ	'rezults'
T	Tn0(3,1)	Λ	'rezults'
T	Tn0(3,2)	Λ	'rezults'
T	Tn0(3,3)	Λ	'rezults'
(Ts0(1,1)<0.5)	Cond0(1,1)	Λ	'rezults'
(Ts0(1,2)<0.5)	Cond0(1,2)	Λ	'rezults'
(Ts0(2,1)<0.5)	Cond0(2,1)	Λ	'rezults'
(Ts0(2,2)<0.5)	Cond0(2,2)	Λ	'rezults'
¬(Ts0(1,1)<0.5)	Cond0(1,1)	Λ	'rezults'
¬(Ts0(1,2)<0.5)	Cond0(1,2)	Λ	'rezults'
¬(Ts0(2,1)<0.5)	Cond0(2,1)		'rezults'
¬(Ts0(2,2)<0.5)	Cond0(2,2)	Λ	'rezults'

Relations						
Name	*ItVal*	*Def*	*Cond*	*Var*	*Value*	*Func*
One	Ø Ø	T	T	a0(1,1)	Λ	INPUT
One	Ø Ø	T	T	a0(1,2)	Λ	INPUT
One	Ø Ø	T	T	a0(1,3)	Λ	INPUT
One	Ø Ø	T	T	a0(2,1)	Λ	INPUT
One	Ø Ø	T	T	a0(2,2)	Λ	INPUT
One	Ø Ø	T	T	a0(2,3)	Λ	INPUT
One	Ø Ø	T	T	a0(3,1)	Λ	INPUT
One	Ø Ø	T	T	a0(3,2)	Λ	INPUT
One	Ø Ø	T	T	a0(3,3)	Λ	INPUT
One	Ø Ø	T	T	b0(1)	Λ	INPUT
One	Ø Ø	T	T	b0(2)	Λ	INPUT
One	Ø Ø	T	T	b0(3)	Λ	INPUT
One	Ø Ø	T	T	T0(1,2)	Λ	INPUT
One	Ø Ø	T	T	T0(1,3)	Λ	INPUT
One	Ø Ø	T	T	T0(2,1)	Λ	INPUT
One	Ø Ø	T	T	T0(3,1)	Λ	INPUT
Two	Ø Ø	T	T	Eps00	Λ	INPUT
Two	Ø Ø	T	T	T00(1)	Λ	INPUT
Two	Ø Ø	T	T	T00(2)	Λ	INPUT
Two	Ø Ø	T	T	T00(3)	Λ	INPUT

The result of performing function **INPUT-DATA-PROCESSING** (*Files*) is filling in relation *InputData*:

Input Data		
Var	*Value*	*File*
a0(1,1)	2.0	'data.dat'
a0(1,2)	3.0	'data.dat'
a0(1,3)	-4.0	'data.dat'
a0(2,1)	3.0	'data.dat'
a0(2,2)	4.0	'data.dat'
a0(2,3)	-1.0	'data.dat'
a0(3,1)	5.0	'data.dat'
a0(3,2)	5.0	'data.dat'
a0(3,3)	5.0	'data.dat'
b0(1)	5.0	'data.dat'

Var	*Value*	*File*
b0(1)	7.0	'data.dat'
b0(1)	9.0	'data.dat'
T0(1,2)	1.1	'norma.dat'
T0(1,2)	2.2	'norma.dat'
T0(1,2)	3.3	'norma.dat'
T0(1,2)	4.4	'norma.dat'
Eps00	1.0E-6	'norma.dat'
T00(1)	-1.1	'norma.dat'
T00(1)	-1.1	'norma.dat'
T00(1)	-1.1	'norma.dat'

The result of performing function **CHECK-INPUT-DATA**(*P*) - filling in the values of attribute *Value* in relation *Input*:

Input		
Var	*Value*	*File*
a0(1,1)	2.0	'data.dat'
a0(1,2)	3.0	'data.dat'
a0(1,3)	-4.0	'data.dat'
a0(2,1)	3.0	'data.dat'
a0(2,2)	4.0	'data.dat'
a0(2,3)	-1.0	'data.dat'
a0(3,1)	5.0	'data.dat'
a0(3,2)	5.0	'data.dat'
a0(3,3)	5.0	'data.dat'
b0(1)	5.0	'data.dat'

Var	*Value*	*File*
b0(1)	7.0	'data.dat'
b0(1)	9.0	'data.dat'
T0(1,2)	1.1	'norma.dat'
T0(1,2)	2.2	'norma.dat'
T0(1,2)	3.3	'norma.dat'
T0(1,2)	4.4	'norma.dat'
Eps00	1.0E-6	'norma.dat'
T00(1)	-1.1	'norma.dat'
T00(1)	-1.1	'norma.dat'
T00(1)	-1.1	'norma.dat'

The result of performing function **CALL-GRAPH-CREATE**(*P*) is designing the graph of functions and parts' calls $G=(V,E)$, V = {**One**,**Two**(**Ts00RESULTTn00**)}, V = (*One*,*Two*). The result of performing function **CHECK-CALL-GRAPH**(*P*) - **T**.

The result of performing function **OPERATORS-TO-RELATIONS**(*o*) - addition to relation *Relations* (*ui*-unique names obtained in the result of using function *uname*(*P*)):

Relations						
Name	*ItVal*	*Def*	*Cond*	*Var*	*Value*	*Func*
One	∅ ∅	T	T	a0(1,1)	Λ	INPUT
One	∅ ∅	T	T	a0(1,2)	Λ	INPUT
One	∅ ∅	T	T	a0(1,3)	Λ	INPUT
One	∅ ∅	T	T	a0(2,1)	Λ	INPUT
One	∅ ∅	T	T	a0(2,2)	Λ	INPUT
One	∅ ∅	T	T	a0(2,3)	Λ	INPUT
One	∅ ∅	T	T	a0(3,1)	Λ	INPUT
One	∅ ∅	T	T	a0(3,2)	Λ	INPUT
One	∅ ∅	T	T	a0(3,3)	Λ	INPUT
One	∅ ∅	T	T	b0(1)	Λ	INPUT
One	∅ ∅	T	T	b0(2)	Λ	INPUT
One	∅ ∅	T	T	b0(3)	Λ	INPUT
One	∅ ∅	T	T	T0(1,2)	Λ	INPUT
One	∅ ∅	T	T	T0(1,3)	Λ	INPUT
One	∅ ∅	T	T	T0(2,1)	Λ	INPUT
One	∅ ∅	T	T	T0(3,1)	Λ	INPUT
Two	∅ ∅	T	T	Eps00	Λ	INPUT
Two	∅ ∅	T	T	T00(1)	Λ	INPUT
Two	∅ ∅	T	T	T00(2)	Λ	INPUT
Two	∅ ∅	T	T	T00(3)	Λ	INPUT
One	∅ ∅	Def(y0(1),u1)	T	x0(1)	Λ	y0(1)+u1
One	∅ ∅	Def(a0(1,1),a0(1,2), a0(1,3),b0(1), b0(2),b0(3))	T	u1	Λ	SUM((oj0)a0(1,j0) * b0(j0))
One	∅ ∅	Def(y0(2),u2)	T	x0(2)	Λ	y0(2)+u2
One	∅ ∅	Def(a0(2,1),a0(2,2), a0(2,3),b0(1), b0(2),b0(3))	T	u2	Λ	SUM((oj0)a0(2,j0) * b0(j0))
One	∅ ∅	Def(y0(3),u3)	T	x0(3)	Λ	y0(3)+u3
One	∅ ∅	Def(a0(3,1),a0(3,2), a0(3,3),b0(1), b0(2),b0(3))	T	u3	Λ	SUM((oj0)a0(3,j0) * b0(j0))
One	∅ ∅	Def(b0(1))	T	y0(1)	Λ	b0(1)
One	∅ ∅	Def(b0(2))	T	y0(2)	Λ	b0(2)
One	∅ ∅	Def(b0(3))	T	y0(3)	Λ	b0(3)
One	∅ ∅	Def(T0(1,2),T0(2,1))	T	T0(2,2)	Λ	T0(1,2)+T0(2,1)
One	∅ ∅	Def(T0(1,3),T0(2,2))	T	T0(2,3)	Λ	T0(1,3)+T0(2,2)
One	∅ ∅	Def(T0(2,2),T0(3,1))	T	T0(3,2)	Λ	T0(2,2)+T0(3,1)
One	∅ ∅	Def(T0(2,3),T0(3,2))	T	T0(3,3)	Λ	T0(2,3)+T0(3,2)
One	∅ ∅	Def(T0(2,2))	T	Ts0(2,2)	Λ	SIN(T0(2,2))
One	∅ ∅	Def(T0(2,3))	T	Ts0(2,3)	Λ	SIN(T0(2,3))
One	∅ ∅	Def(T0(3,2))	T	Ts0(3,2)	Λ	SIN(T0(3,2))
One	∅ ∅	Def(T0(3,3))	T	Ts0(3,3)	Λ	SIN(T0(3,3))
One	∅ ∅	T	T	Ts0(1,1)	Λ	SIN(1+1)
One	∅ ∅	T	T	Ts0(1,2)	Λ	SIN(1+2)
One	∅ ∅	T	T	Ts0(1,3)	Λ	SIN(1+3)
One	∅ ∅	T	T	Ts0(2,1)	Λ	SIN(2-1)
One	∅ ∅	T	T	Ts0(3,1)	Λ	SIN(3-1)
One	∅ ∅	Def(Ts0(1,1), Ts0(2,1), Ts0(3,1))	T	Tn0(1,1), Tn0(1,2), Tn0(1,3)	Λ Λ Λ	Two(Ts0 ON Oij0/j0=1 REZULT Tn0 ON Oij0/i0=1)

One	∅ ∅	Def(Ts0(1,2), Ts0(2,2), Ts0(3,2))	T	Tn0(2,1), Tn0(2,2), Tn0(2,3)	Λ Λ Λ	Two(Ts0 ON Oij0/j0=2 REZULT Tn0 ON Oij0/i0=2)
One	∅ ∅	Def(Ts0(1,3), Ts0(2,3), Ts0(3,3))	T	Tn0(3,1), Tn0(3,2), Tn0(3,3)	Λ Λ Λ	Two(Ts0 ON Oij0/j0=3 REZULT Tn0 ON Oij0/i0=3)
One	∅ ∅	Def(Tn0(1,1), Ts0(1,1))	Ts0(1,1)<0.5	Cond0(1,1)	Λ	Tn0(1,1)
One	∅ ∅	Def(Tn0(1,2), Ts0(1,2))	Ts0(1,2)<0.5	Cond0(1,2)	Λ	Tn0(1,2)
One	∅ ∅	Def(Tn0(2,1), Ts0(2,1))	Ts0(2,1)<0.5	Cond0(2,1)	Λ	Tn0(2,1)
One	∅ ∅	Def(Tn0(2,2), Ts0(2,2))	Ts0(2,2)<0.5	Cond0(2,2)	Λ	Tn0(2,2)
One	∅ ∅	Def(Ts0(1,1))	¬(Ts0(1,1)<0.5)	Cond0(1,1)	Λ	Ts0(1,1)
One	∅ ∅	Def(Ts0(1,1))	¬(Ts0(1,2)<0.5)	Cond0(1,2)	Λ	Ts0(1,1)
One	∅ ∅	Def(Ts0(1,1))	¬(Ts0(2,1)<0.5)	Cond0(2,1)	Λ	Ts0(1,1)
One	∅ ∅	Def(Ts0(1,1))	¬(Ts0(2,2)<0.5)	Cond0(2,2)	Λ	Ts0(1,1)
Two	N00 0	Def(Ts00(1))	T	Tn00(1,N00)	Λ	Ts00(1)
Two	N00 0	Def(Ts00(2))	T	Tn00(2,N00)	Λ	Ts00(2)
Two	N00 0	Def(Ts00(3))	T	Tn00(3,N00)	Λ	Ts00(3)
Two	N00 1	Def(Tn00(1,N00-1))	T	Tn00(1,N00)	Λ	Tn00(1,N00-1)/3
Two	N00 1	Def(Tn00(2,N00-1))	T	Tn00(2,N00)	Λ	Tn00(2,N00-1)/3
Two	N00 1	Def(Tn00(3,N00-1))	T	Tn00(3,N00)	Λ	Tn00(3,N00-1)/3
Two	N00 1	Def(Tn00(3,N00), Eps00)	ABS(Tn00(3,N00)) < Eps00	∅	∅	∅

Design of abstract program $A(P)$ is finally finished. Now here is a brief description of *AM*-machine working by this program assuming that *NonDeterm=ArgDefined* and *NonDetermOut = OutDefined*. Thus to eliminate non-determination of *AM*-machine and not to specify all the variants of possible computations we have to demonstrate semantics of natural(ideal) parallelism.

Initialization.

Checking **CHECK-REASSIGNMENT**(*Relations*) gives T after that function **CALL-PART**(**One**) is performed invoking tuples **F**(*Relations*, *Name* = **One**).

Stage of interpretation 1.

a) Relation *NonDeterm=ArgDefined* determines computation variables a0(1,1),a0(1,2), a0(1,3), a0(2,1), a0(2,2), a0(2,3), a0(3,1), a0(3,2), a0(3,3), b0(1), b0(2), b0(3), T0(1,2), T0(1,3), T0(2,1), T0(3,1), Ts0(1,1), Ts0(1,2), Ts0(1,3), Ts0(2,1), Ts0(3,1). The values of these variables are computed corresponding to function *func* set in the field of attribute *Func*.

b) **CHECK-STOP** = *nextAMstep*

Stage of interpretation 2.

a) Relation *ArgDefined* determines the computation of variables u1, u2, u3, y0(1), y0(2), y0(3), T0(2,2), {Tn0(1,1), Tn0(1,2), Tn0(1,3)}. The values of these variables are computed. Computation of {Tn0(1,1), Tn0(1,2), Tn0(1,3)} engenders call of function

CALL-PART(**Two**(Ts0 ON Oij0/j0=1 **RESULT** Tn0 ON Oij0/i0=1))

and invoking the tuples of part **Two**.

An extract of relation *Relations* is -obtained as the result of invoking the tuples of part **Two**.

ARelations						
Name	*ItVal*	*Def*	*Cond*	*Var*	*Value*	*Func*
Two	∅ ∅	T	T	Eps00	Λ	**INPUT**
Two	∅ ∅	T	T	T00(1)	Λ	**INPUT**
Two	∅ ∅	T	T	T00(2)	Λ	**INPUT**
Two	∅ ∅	T	T	T00(3)	Λ	**INPUT**
Two	∅ ∅	Def(Ts0(1,1))	T	u4	Λ	Ts0(1,1)
Two	∅ ∅	Def(Ts0(2,1))	T	u5	Λ	Ts0(2,1)
Two	∅ ∅	Def(Ts0(3,1))	T	u6	Λ	Ts0(3,1)
Two	∅ ∅	Def(u4)	T	Ts00(1)	Λ	u4
Two	∅ ∅	Def(u5)	T	Ts00(2)	Λ	u5
Two	∅ ∅	Def(u6)	T	Ts00(3)	Λ	u6
Two	N00 0	Def(Ts00(1))	T	Tn00(1,0)	Λ	Ts00(1)
Two	N00 0	Def(Ts00(2))	T	Tn00(2,0)	Λ	Ts00(2)
Two	N00 0	Def(Ts00(3))	T	Tn00(3,0)	Λ	Ts00(3)
Two	N00 1	Def(Tn00(1,N00-1))	T	Tn00(1,1)	Λ	Tn00(1,0)/3
Two	N00 1	Def(Tn00(2,N00-1))	T	Tn00(2,1)	Λ	Tn00(2,0)/3
Two	N00 1	Def(Tn00(3,N00-1))	T	Tn00(3,1)	Λ	Tn00(3,0)/3
Two	N00 1	Def(Tn00(3,1), Eps00)	ABS(Tn00(3,1)) < Eps00	∅	∅	∅
Two	∅ ∅	Def(Tn00(1,N00))	T	u7	Λ	Tn00(1,N00)
Two	∅ ∅	Def(Tn00(2,N00))	T	u8	Λ	Tn00(2,N00)
Two	∅ ∅	Def(Tn00(3,N00))	T	u9	Λ	Tn00(3,N00)
Two	∅ ∅	Def(u7)	T	Tn00(1,1)	Λ	u7
Two	∅ ∅	Def(u8)	T	Tn00(1,2)	Λ	u8
Two	∅ ∅	Def(u9)	T	Tn00(1,3)	Λ	u9

b) **CHECK-STOP** =*nextAMstep*

Stage of interpretation 3

a) relation *ArgDefined* determines the computation of variables x0(1), x0(2), x0(3), T0(2,3), T0(3,2), Ts0(2,2), u4, u5, u6, Eps00, T00(1), T00(2), T00(3). The values of these variables are computed.

b) Relation *OutDefined* determines variables x0(1), x0(2), x0(3),which values has been computed. Output of these values is carried out into file 'results'.

c) **CHECK-STOP** =*nextAMstep*

Stage of interpretation 4.

a) Relation *ArgDefined* determines the computation of variables T0(3,3), Ts0(2,3), Ts0(3,2) u4, u5, u6, Ts00(1), Ts00(2), Ts00(3). The values of these variables are computed.

b) Relation *OutDefined* determines variable T0(3,3) which value has been computed. Output of these values is carried out into file 'results'.

c) **CHECK-STOP** =*nextAMstep*

Stage of interpretation 5.

a) Relation *ArgDefined* determines the computation of Ts0(3,3), {Tn0(2,1), Tn0(2,2), Tn0(2,3)}, Tn00(1,0), Tn00(2,0), Tn00(3,0). The values of these variables are computed. Computation of {Tn0(1,1), Tn0(1,2), Tn0(1,3)} engenders call of function

CALL-PART(**Two**(Ts0 ON Oij0/j0=2 **RESULT** Tn0 ON Oij0/i0=2))

and invoking the tuples of part **Two** (this process and the computations determined by it aren't described to make the text shorter).

c) **CHECK-STOP** =*nextAMstep*

Stage of interpretation 6.

a) Relation *ArgDefined* determines the computation Tn00(1,1), Tn00(2,1), Tn00(3,1), {Tn0(3,1), Tn0(3,2), Tn0(3,3)}. The values of these variables are computed. Computation of {Tn0(3,1), Tn0(3,2), Tn0(3,3)}. engenders call of function

CALL-PART(Two(Ts0 ON Oij0/j0=3 **RESULT** Tn0 ON Oij0/i0=3))
and invoking the tuples of part **Two** (this process and the computations determined by it aren't described to make the text shorter).

Exit condition of iteration ABS(Tn00(3,1))<Eps00 is checked and either the next step to compute Tn00(1,1), Tn00(2,1), Tn00(3,1) is done or passing the result is performed. Let's assume that iteration is finished.

c) **CHECK-STOP** =*nextAMstep*

Stage of interpretation 7.

a) Relation *ArgDefined* determines the computation of u7, u8, u9. The values of these variables are computed.

c) **CHECK-STOP** =*nextAMstep*

Stage of interpretation 8.

a) Relation *ArgDefined* determines the computation of Tn0(1,1), Tn0(1,2), Tn0(1,3). The values of these variables are computed.

b) Relation *OutDefined* determines variables Tn0(1,1), Tn0(1,2), Tn0(1,3) which values have been computed. Output of these values is carried out into file 'results'.

c) **CHECK-STOP** =*nextAMstep*

Stage of interpretation 9.

a) Relation *ArgDefined* determines the computation of

Cond0(1,1) (if Ts0(1,1)<0.5),
Cond0(1,2) (ifTs0(1,2)<0.5),
Cond0(1,1) (if ¬(Ts0(1,1)<0.5)),
Cond0(1,2) (if ¬(Ts0(1,2)<0.5)).

The tuples determining the computation of Cond0(1,1) (if ¬(Ts0(1,1)<0.5)), Cond0(1,2) (if ¬(Ts0(1,2)<0.5)) are eliminated from *ArgDefined* and *ARelations* as the condition *Cond*=**F**.

The values of the variables Cond0(1,1) (if Ts0(1,1)<0.5), Cond0(1,2) (ifTs0(1,2)<0.5) are computed.

b) Relation *OutDefined* defines variables Cond0(1,1), Cond0(1,2) which values have been computed. The tuples describing output Cond0(1,1) (if ¬(Ts0(1,1)<0.5)), Cond0(1,2) (if ¬(Ts0(1,2)<0.5)) are eliminated from *OutDefined* as condition *Cond*=**F**. Output of the values Cond0(1,1) (if Ts0(1,1)<0.5), Cond0(1,2) (ifTs0(1,2)<0.5) is cariied out into file "results".

c) **CHECK-STOP** =*nextAMstep*

Stage of interpretation ...

CHECK-STOP =*nextAMstep*

2.10 Definite work of *AM*-machine

The value of variable $X \in ARelations(Var)$ are computable if it has no information dependencies or the values of all variables $Y_1,...,Y_n$,from which X_1 has information dependencies have been computed.

$$\mathbf{COMPUTABLE}(X) \sim$$
$$\exists\alpha(\alpha \in ARelations \wedge \alpha = \langle name \circ itval \circ def \circ cond \circ var \circ value \circ func \rangle$$
$$\wedge X = var \wedge \mathbf{LOG\text{-}EXPR}(def) = \mathbf{T} \wedge \mathbf{LOG\text{-}EXPR}(cond) = \mathbf{T} \wedge value \neq \Lambda)$$

(In this definition function **LOG-EXPR**(e) computes the value of logical expression e).

Computation of function *func* = **F**(*ARelations, Var* = X)(*Func*) if **COMPUTABLE**(X) ≡ **T** is called *computation of the* value of variable $X \in ARelations(Var)$.

$$\mathbf{COMPUTATION}(X) \sim \exists\alpha(\alpha \in ARelations \wedge X \propto \alpha \wedge \mathbf{COMPUTABLE}(X))$$
$$\rightarrow \mathbf{COMPUTE}(\alpha))$$

Variable X is determined definitely if program P satisfies the condition of single-assignment and $X \in ARelations(Var)$.

DEF-ONE-TO-ONE(X) ~ **CHECK-REASSIGNMENT**$(Relations)$ $\wedge X \in ARelations(Var)$

Program P is *correct* if it is semantically correct and the work of AM-machine is terminated.

Theorem. *All the values of the variables are determined definitely for correct program P.*

Proof.

Work of AM-machine finishes in state *stop* for correct program so the number of interpretation stages is finite and according to definition of state *stop* the values of all the variables which are to be computed are computed.

The condition of single-assignment is fulfilled for correct program as function **CHECK-REASSIGNMENT** is done by AM-machine. Thus all the variables of program P are determined definitely in accordance with the definition of definite determination.

The arbitrariness of choosing set $NonDeterm \subseteq ArgDefined$ at each stage of interpretation and computation of the variables' values from $Nondeterm$ don't breake definite determination of the variables.

2.11 Notes to semantics' specification

1) The specification of the NORMA language's semantics given above is a shortened specification. Thus some functions of low level (e.g. **HEAD**(l), **TAIL**(l), l- list) and also the number of functions fixing semantic errors aren't defined. It is done to make the text of report shorter. Descriptions of functions fixing semantic errors are to be given in working out of NORMA translator's realisation .

2) The specification of the NORMA semantics isn't to be considered as the complete one. Some more details and modifications are possible to be added in the process of further researching. It may specify the rules of the NORMA language interpretation. Particularly the methods of specifying computational environment where Norma program can be carried out are under consideration now. The result of this research may give more details to the NORMA language.

3) Definition of an abstract machine AM-machine is a theoretical model which is the base for working out algorithms of Norma program translation. This problem isn't easy as the algorithms must be effective by time and memory, consider the peculiarities of the target computer and be equivalent to the notion of AM-machine. Obtained theoretical results (e.g. [3]) and worked out experimental version of the NORMA language's translator allow to consider this problem practically solvable.

4) Approach used in the semantics' specification is based on the representation of information dependencies between the variables from NORMA program by means of relations from relational algebra and the rules of interpreting these relations. This approach may be used for the specification of the semantics of the languages similar to NORMA which may be used for solving problems in other application domains, e.g. in the domain of relational data bases. To specify the semantics of the languages of considered type one may use other approaches based on the data driven computation model, e.g. data flow model.

References.

[1] **E.Z. Lubimsky**. The problems of programming's automation. Bulletin of Academy of Sc. USSR, 8,1960.(in Russian)

[2] **I.B. Zadykhailo**. Organising loop process of computations on parameter record of special type. Journal of calcul. math. and math. physics, 3, 2(1963), p.337-357. (in Russian)

[3] **A.N. Andrianov**. The synthesis of loop computations in the NORMA language. Preprint of Keldysh Inst. of Appl. Math., Russian Academy of Science, 171(1986), p.28.

[4] **L. Lamport**. The parallel execution of DO loops. Communications ACM, 17, 2(1974), pp.83-93.

[5] **G.L. Steele.** High Performance Fortran: status report. Workshop on Languages, Compilers and Run-Time Enviroments for Distributed Memory Multiprocessors, Boulder, CO, 30 Sept.-2 Oct. 1992, reprint in SIGPLAN NOTICES 28 1 (1993) pp.1-4.

[6] **P.H. Welch, G.R.R. Justo**. On the serialisation of parallel programs. WoTUG-14, Longhborough University, Sept.1991.

[7] **A.N. Andrianov, A.B. Bugerja, K.N. Efimkin, I.B. Zadykhailo**. The specification of the NORMA language. Draft Standard Preprint of Keldysh Ins. of Appl. Math. , Russian Academy of Sc., 120(1995), pp.1-50.

[8] *Semantics of the programming languages*. Mir, Moscow, 1980.(in Russian)

[9] **A. Ollongren**. *Definition of programming languages by interpreting automata*. Academic Press, (London, New York, San Francisco), 1974.

[10] *E. Ozkarahan*. Database machines and database management. Prentice-Hall, Englewood Cliffs,1986.

[11] **E.F. Godd**. A relational model of data for large shared data banks. Communications of ACM, 6, 13(1970).

[12] **D. Gries**. Compiler construction for digital computers. John Wiley, (New York, London, Sydney, Toronto), 1971.

Russian Academy of Science
Keldysh Institute of Applied Mathematics

Andrianov A.N., Bugerya A.B., Efimkin K.N., Zadykhailo I.B.

NORMA

Language specification. Draft copy 1.22

Moscow 1996

Andrianov A.N., Bugerya A.B., Efimkin K.N., Zadykhailo I.B.

NORMA. Language specification. Draft copy 1-22.

Abstract.

The NORMA language is a tool aimed at automatic solution of the mathematical physics problems on parallel computer systems.

The aim of the NORMA language is to eliminate the programming phase which is necessary to pass from computational formulae derived by an application specialist to a computer program. There is no essential difference between computational formulae and NORMA program structures - these formulae are an input for the NORMA translating system.

In fact NORMA program is a nonprocedural specification of problems to be solved. The mathematical problems connected with the synthesis of output program are solvable in the case of the NORMA language.

Draft specification of the NORMA language is given.

Contents

Introduction.

NORMA is a declarative language aimed at computation tasks' specifications. Translator of the language has besides traditional functions of semantic and syntax analysis a function of output program

synthesis during the translation .In other words the mode and the order of computations depended on target computer language and architecture are arranged.

The principal ideas of automated program design based on task's specification were formulated by I.B. Zadykhailo [1] even in 1963. Their further development gave birth to NORMA and several versions of the translator [2-14].Some examples of NORMA practical usage are described in [15,16].

Authors consider the approach used in Norma design and realisation useful for creation of new generation languages. The development of parallel computers enhances the value of this method.

This paper describes syntax and semantics of the NORMA language.

1. Technical background.

The idea of the NORMA language was produced by applied mathematicians from KIAM (Keldysh Institute of Applied mathematics) RAS (Russian Academy of Science).It was an attempt of automated program design based on the jobs (application of numerical methods to physical problems solution) prepared for further programming.

It was 1960 when the first works on this theme appeared in our institute (e.g.,[17]).The method of writing jobs in this work was called "parameter record". Later on such methods had the name "non-procedure specifications" and afterwards they were called "declarative specifications". The languages used these methods are of specification type where the rules of data computations must be defined but the order of computations may be arbitrary (e.g. in parallel or sequential mode) according to the rules.

Many researches on programming theory and practical results proved the correctness of the chosen path. Some corroborating aspects are given below.

2. Aims.

Practical value of application software design depends on the degree of:

1) *automation* of design process,

2) *portability* and *reliability* of the output program,

3)convenience for the specialist in particular *application domain.*

The problems of providing all these items has been solving for the whole period of computer usage but there have been no universal and perfect methods for their solution yet. It can be explained by complicity of these problem and very fast development of the programming as a branch of science (new application domains and new hardware appear very quickly).

Progress in computer technology and architecture provides ample computational potentialities. But taking into consideration [1-3] requirements their perfect practical application has many difficulties. Thus when multiprocessor computers appeared the question of new application software design was raised.

Probably application programmer wants to know as little as possible about the architecture of a particular computer but to use its resources as much as he could. He'd like to have convenient tools for test and debugging but no problems with programs' portability. To work using generic methods and terms is also very important for application programmer. He needn't know questions which are alien to him [compiler's peculiarities, programming languages, system library organization and the questions of its effective usage, etc.].

The NORMA language and the approach to application program design is an attempt to take into consideration [1-3] requirements. The authors of this approach realize the difficulties of the fundamental task solution and the fact that universal and perfect way to this solution couldn't be found immediately Though we can point out some goals of these approach even now.

3. The goals of NORMA usage.

High level of automation of the application design process.

Initial equations are formalized only to make input from keyboard easy. It is important that the issue of the formulae isn't changed. Thus the programming phase is skipped and further design of the program is carried out automatically

Reliability and portability of the programs.

If the method of the task solution is right and the formulae are written correctly an output program will be also correct (because the process of programming is automatic).

We have to underline that the *strict* specification of the task solution process on the *application domain* level is very important. This level is the most reliable because it deals only with the scheme of the computations and doesn't depend on the optimizations which can be realized during programming phase. It is evident that if we propose the language for the strict specification of the task solution process in the terms of application domain automatically designed program based on this specification will have the degree of reliability corresponding to the one of the translator and the translator's optimization level.

NORMA used only the terms of application domain associated with the grid-based mathematical physics tasks solutions. Simplicity and reliability of the programs are provided by eliminating of the programming phase and required only the knowledge of application domain terms and their correct usage.

Synthesising translator from NORMA allows for computer architecture peculiarities. It provides portability of the application programs.

Method of parallel program design.

Generic mathematical formulae usage in NORMA has great possibilities for problem optimization in all cases including realization on parallel systems.

The specification of the task solution keeps its natural parallelism. This specification doesn't contain any adaptation constraints to computer architecture, programming language peculiarities or other requirements. It is "clear" specification of task solution which restricts only computation order according to the computation scheme content (informatic dependencies between variables).There is no term "memory" for storing different values in different periods of time. It simplifies natural parallelism revealing algorithms design and creation of the target program.

In the cases where computation order is important (e.g. . it affects the precision of computations, rate of convergence, etc.) supplementary notions may be used (they are interpreted in a special way).

The development of the language makes its life longer. NORMA is rather young and capable.

Norma language specification.

1. Application domain and main characteristics.

The NORMA language is a specialized language applied to numerical - based specification of the mathematical physics problems solutions. First it was directed towards solving mathematical physics ' problems by means of grid difference method solutions. But later practice shows us that NORMA application domain could be more extensive.

Originally NORMA could be deciphered as : Non-procedure Specification of the Difference Algorithm Models. Nowadays we decode this abbreviation as NORMAl level of computer - mathematician communication. Mathematician's formulae are input almost directly into the computer system.

NORMA requires no information about computation order and ways of computation process organization. The order of the sentences may be arbitrary (informatic connections are revealed_during the translation).

Value can be assigned to any variable in NORMA only once. This feature characterizes the level of the NORMA language. There is no such terms as "memory", side effect, assignment statement, control operators in NORMA.

These characteristics and some other constraints (first on the form of index statements and ways of index domain specifications) substantiate solvability of output program synthesis problem [6,9]. Generic solution of this problem has substantial mathematical difficulties (the task may be NP-complete or undecidable).On the other hand researches on NORMA applications and design show that the constraints are acceptable practically [15,16].

2 Syntax notation.

Extended Backus - Naur form is used in the given syntax notation.

$\{A\}^*$, $\{A\}^+$, $\{A_1,\ldots A_n\}$, [A] symbols mean

$$\{A\}^* \quad ::= \quad \varnothing \mid A \mid A\,A \ \ldots$$

$$\{A\}^+ \quad ::= \quad A \mid A\,A \ \ldots$$

$\{A_{1,\ldots,} A_n$ ::= $A_1 | \ldots | A_n$

$\}$ ::= $\varnothing \mid A$,

[A]

where A - any language object ,

$\varnothing$ - empty,

| - alternative choose,

... etc.

Syntax notions are written in italics, but words and symbols of common usage are printed in a usual way. As a rule, alternative constructions are arranged in columns (each construction on a separate line).

Sometimes half-underlined syntax constructions are used, e.g. *name - set* .Syntactically this name is identical to symbol *name* but underlined part of the construction has additional semantic information.

Symbol list-*element* substitutes non-empty list of *elements* enumerated by comma:

list-*element* :

element { ,*element* }*

The definition of the element is given in every particular case.

Usually syntactical rules are given first; commentaries, semantic explanations, examples, etc. follow.

3. Principal language elements.

3.1 Lexical rules.

Symbols are a base for all the constructions. they are original language elements. The number of principal symbols is fixed and used for the design of any language constructions. The number of supplementary symbols isn't fixed and settled by the type of computer equipment. They are used for symbolic constants formation of symbolic constants and data representation for medium.

3.1.1. *Principal symbols.*

principal-symbol :

letter

digit

special-symbol

letter :

{ A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z }

{ a,b,c,d,e,f,g,h,i,j k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}

digit :

{ 0,1,2,3,4,5,6,7,8,9 }

special-symbol :

special-symbol-not-apostrophe

apostrophe

special-symbol-not-apostrophe :

{ *space* ,=,>,>,+,-,*,/,(,),[,],!,?,,,,.,#}

apostrophe :

'

space :

Space has no graphical representation.

3.1.2. *Commentaries.*

The line which begins from the sign "!" or the part of line which follows the sign "!" is a commentary.

3.1.3. *Tokens.*

There are 5 classes of tokens: *identificators, key - words, constants, operation signs, delimiters.*

Spaces, LF, commentaries are considered tokens' delimiters and have no influence on the program semantics.

3.1.4. *Identificators.*

identificator :

letter { { *digit, letter* } }*

The letters of upper register differ from the ones of low register. Identificators can have any length. The number of the meaningful symbols is defined in the specification of the translator's source language (usually first 6 symbols are meaningful).

3.1.5. *Key words.*

These key-words are used in the language:

MAIN PART	**VARIABLE**	**ITERATION**
PART	**DEFINED ON**	**BOUNDARY**

FUNCTION	**REAL**	**END BOUNDARY**
BEGIN	**INTEGER**	**INITIAL**
END PART	**DOUBLE**	**END INITIAL**
RESULT	**MACRO INDEX**	**EXIT WHEN**
EXTERNAL FUNCTION	**DOMAIN**	**END ITERATION**
	PARAMETERS	
EXTERNAL PART	**INPUT**	**FOR**
DISTRIBUTION INDEX	**OUTPUT**	**ASSUME**
INDEX	**ON**	**COMPUTE**

3.1.6. *Constants.*

constant :

arithm-constant

string

arithm-constant:

int-constant

real-constant

double-constant

int-constant :

{ *digit* }$^{+}$

real-constant :

body [**E** [{ +,- }] *power*]

double-constant :

body [**D** [{ +,- }] *power*]

body :

int-constant .int-constant

power

int-constant

string

' { { *symbol-not-apostrophe*, ' ' } }$^{+}$'

symbol-not-apostrophe:

letter

digit

special- symbol-not-apostrophe

supplementary-symbol

The rules of writing constants are similar to the ones in the Fortran language.

The type and the value of the constant is defined by the way of its representation. These are examples of arithmetical constants :

101 999 0.1 1.0

10.5E-6 1.0E+7 1.0E7 0.1D-8 15.333D4

These are examples of the strings :

'Hello, world!'

'Îá ''åì òåëà = '

3.1.7. *Signs of operations.*

operation :

{real-operation, log-operation, arithm-operation }

real-operation:

{ =,>,<,<=,>=,<>,>< }

real-except-not-equal-operation :

{ =,>,<,>=,<= }

log-operation :

{ **AND, OR, NOT** }

arithm-operation :

{ +,-,*,/,** }

Symbols >= and <= are used for comparisons ≥ and ≤ , <> and >< symbols - for ≠ .

3.1.8. *Delimiters.*

delimiter :

{ *space*,/,(,),[,],!,?,',,,,,# }

3.2. Names.

name :

identificator

list-n*ame :*

*name { ,name }**

Name is an identificator. It has a particular meaning and defines a particular program's object (e.g. variable, domain, part of the program). There can't be two definitions of the objects which have the same name.

The rule of name localization is true for every program element (part or function). The names declared in the program element are localized in it. There is no notion " global variable " in this language. The

names of the functions or the parts if they are actual or formal parameters of other parts or functions are indicated in the *external names declaration* **EXTERNAL FUNCTION** or **EXTERNAL PART** [see 5.16].

Half - underlined syntactical symbols e.g. *name-set* are syntactically equivalent to symbol *name*. Underlined part of the construction has additional semantic information.

Symbol **list-***name* substitutes non-empty list of the *names* enumerated by comma.

3.3 Basic data types.

There are basic data types specified in the language:

integer

real

double

logical

string

Value from the set of integers is assigned to the element of *integer* data type. There are *arithmetical* and *comparison* operations for integer type elements.

Value from the set of real numbers is assigned to the element of *real* data type. There are *arithmetical* and *comparison* operations for real type elements.

Value from the set of real numbers is assigned to the element of *double* data type and it allows to use double precious *real* numbers. There are *arithmetical* and comparison operations for double type elements.

There are variables of *integer, real* and *double* type in the language.

Value "true" and "false" is assigned to the element of *logical* data type. It is used for definition of condition. There are *logical* operations for logical type elements. There are no variables of logical type in the language.

The element of *string* data type is a set of symbols from *principal symbol* set and *supplementary symbol* set. There are only constants but no variables and operations of this type.

3.4. Expressions.

3.4.1. Arithmetical expressions.

arithm-expression :

*[{ +, - }] term { arithm-operation term }**

term :

arihm-constant

call-function

name-index

name-domain-parameter

name-scalar

variable-on-domain

(arithm-expression)

variable-on-domain :

name-variable-on-domain[[list-index-expression]]

const-expression :

[{ +,- }] const-expression-without-sign

const-expression-without-sign :

*integer-term { arithm-operation integer-term }**

integer-term :

int-constant

name-domain-parameter

(const-expression)

index-expression :

name-index [{ +,- }] const-expression-without-sign]

name-index = const-expression-without-sign

name-index = name-index [{ +,- }] const-expression-without-sign]

name-index-construction

The result of the *index expression* computation in a fixed point T of the domain (5.1.1.) is coordinates of the point T'. These coordinates can be defined according to the rules :

(1) all the names of the index constructions included into the *index expression* are substituted to the corresponding lists of index expressions from **MACRO INDEX** declaration ;

(2) indexes are substituted to the *constant expressions*, values of T coordinates (taking into account index displacement if it is given) or values of the other given indexes in the point T.

For example, the list of index expressions **[MyIJ,k=7,l=i+4]** for the point T with i=9,j=15,k=3,l=37 (if we have the declaration **MACRO INDEX MyIJ [i-1,j]**) gives us the value of the point T from index domain with i=8,j=15,k=7,l=13 coordinates.

The order of *arithmetical expression* computation is :

(1) *index expressions* computation;

(2) finding arguments (actual parameters) of the functions (including determination of reduction functions operation domain [5.2.3];

(3) carrying-out in-brackets (and) operations: if there are several brackets of the one level, they are examined from the left to the right;

(4) carrying-out raising to the ** power operations: if there are several operations of the one level, they are examined from the right to the left;

(5) carrying-out of multiplication and division * and / operations; if there are several operations of the one level, they are examined from the left to the right;

(6) carrying-out of sum and difference + and - operations; if there are several operations of the one level, they are examined from the left to the right.

The type of arithmetical expression result is defined by the types of operation results. The type of operation result can be :

(1) integer, if both operands are integer :

(2) double, if even one operand is double ;

(3) real in any other cases.

Constant expression is a particular case of *arithmetical expression*. The examples of arithmetical expressions :

AL12[J=2*M-1]+STEPLP*JJ

Q+1.0D5/((R[Q-1]-R)*(R[Q-1]-R[Q+1]))

CN*TRAP(STEP, 2*MV+1, MMR ON POLL)

3.4.2. *Conditional expressions.*

log-expression :

[**NOT**] *logic-term* { *log-operation logic-term* }*

logic-term :

comparison

(*log-expression*)

comparison :

arithm-expression comp-operation arithm-expression

condition-on-domain :

log-expression

condition-on-index :

name-index rel-except-not-equal-operation

name-index [{ +,- }*const-expression-without-sign*]

name-index rel-except-not-equal-operation const-expression-without-sign

The order of operations for *logical expression* computation :

(1) carrying out of arithmetical expressions computation;

(2) carrying out of in-brackets (and) operations; if there are several brackets of the one level, they are examined from the left to the right.

(3) carrying out of comparisons =,>,<,>=,<=,<>,>< ; if there are several operations of the one level, they are examined from the left to the right.

(4) carrying out of logical negation **NOT** operations; if there are several operations of the one level, they are examined from the left to the right.

(5) carrying out of logical multiplication **AND** operations; if there are several operations of the one level, they are examined from the left to the right.

(6) carrying out of logical sum **OR** operations; if there are several operations of the one level, they are examined from the left to the right.

There are examples of logical expressions:

ABS(MOD-X1+Y1)<=1.0D-20

NOT (II=(2*N+1)*(2*NV+1) AND NU=0)

I=J+1

4. Program structure.

program :

(part)$^{+}$

part :

main-part

simple-part

part-function

main-part :

MAIN PART *name-part*. *declaration-of-part*

simple-part :

PART *name-simple-part . declaration-of-part*

part-function :

FUNCTION *name-function [type-function] . declaration-of-function*

declaration-of-part :

formal-parameters-of-part **BEGIN** *body-of-part* **END PART**

formal-parameters-of-part :

list-*name* [**RESULT** **list-***name*]

declaration-of-function ;

formal-parameters-of-function **BEGIN** *body-of-part* **END PART**

formal-parameters-of-function :

list-*name*

body-of-part :

*{element-of-part}**

element-of-part :

declaration .

operator .

iteration .

Program in NORMA consists of one or several parts. The parts may be of three types - main part, simple part and part-function (corresponding to key-words **MAIN PART**, **PART**, **FUNCTION**). Parts can call each other by name and communicate data with help of formal and actual parameters or through external files with help of **INPUT** and **OUTPUT** declarations.

Localization rule is true for each part: the names declared in the part are localized in it. There are no global variables in the language.

The main part must be in the NORMA program and be the only one (it has no formal parameters. Main part calls and recursive calls are prohibited.

There is name of the part, text-commentary (if you need) and the list of formal parameters in the part header . Formal parameters must be declared in the body of part with help of *declaration on domain, scalar declaration, domain's parameters declaration* and *declaration of external.*

Parameters-variables declared before the key-word **RESULT** in the list of formal parameters are initial data for computations specified in the part, the parameters declared after are the results of computations. One and the same parameter can't be initial and result at the same time (it causes reassignment which is prohibited in NORMA). The key-word **RESULT** isn't used in part-function (the result of computations is connected with the name and the type of function).

There are definitions, operators and iterations in the body of part. In general their order is arbitrary, possible constraints are defined in translator's source language description).

It is an example of main part header :

```
MAIN PART Linear .     ! Variant of May 25 1992 :
!  ======= linear approximation ========
!  the case of axial symmetry for M=1 ;
!  computation is being done on Beta angle for the point I=0 ;
!  further for I=1,2*N values
!     are being extended on symmetry for V,FI,DAVL .
BEGIN
body of part
END PART
```

It is an example of simple part header ;

```
PART IntFKP .
!  Computation of integral from FKP function
!  with interpolation of integral function
  BETA,ALPHA,   ! - node points
  ALL,ALC,ALR.  ! - points for interpolation
  STEP,  ! - integration step
  FKP    ! - name of integral function
         ! - result IntResult
  RESULT      IntResult
BEGIN
EXTERNAL FUNCTION  FKP  DOUBLE .
VARIABLE ALPHA,BETA,ALL,ALK,ALR,STEP,IntResult DOUBLE .
body of part
END PART
```

5. NORMA constructions.

We can divide Norma constructions into *declarations*, defining program objects (e.g. domains, indexes, variables) and *constructions for computation rules specification*.

5.1 *Declarations*

declaration :

declaration-of-domain

declaration-of-domain-indexes

declaration-of-scalar-variables

declaration-of-variables-on-domains

declaration-of-index-constructions

declaration-of-distribution-indexes

declaration-of-domain-parameters

declaration-of-input

declaration-of-output

declaration-of-external

Objects which can be declared in NORMA are: domains, scalar variables (scalars), variables specified on grid, index constructions, distribution indexes, domain parameters, input and output variables, names of external functions and parts.

5.1.1 *Declarations of domains.*

declaration-of-domain :

declaration-of-unconditional-domain

declaration-of-conditional-domain

declaration-of-unconditional-domain

declaration-of-rectangular-domain

declaration-of-diagonal-domain

domain :

new-domain-without-name

name-domain

unconditional-domain :

new-domain-without-name

name-unconditional-domain

name-domain :

name-unconditional-domain

name-conditional-domain

name-unconditional-domain :

name-rectangular-domain

name-diagonal-domain

The notion domain was introduced in NORMA for index space representation. Domain is a complex of integers numbers sets $(i_1,....,i_n)$, $n>0$, $i_j>0$, $j=1,....n$. each set gives coordinates of the point from n-dimensional space. Unique name - *index name* is connected with one of the directions (coordinate axes) of the n-dimensional space.

The domain defines coordinates values of index space points, but not the values of the calculated variables in these points. For example if you need to calculate the value of the variable $Y_{i,j}$, $i,j=1,....,n$ on some grid $X_{i,j}$, $i,j=1,....n$, which was introduced during task solution (e.g. by formula $X_{i,j}=F(h,i,j)$, F- given function, h - given parameter) you should:

(1) define the domain consists of points $i,j = 1,...n$;

(2) define variables X and Y;

(3) set the rule of grid $X_{i,j}$ computation : $X_{i,j} = F(h,i,j)$ and the rule of $Y_{i,j}$ values computation $Y_{i,j} = G(X_{i,j})$ (F,G,h are considered to be given somehow).

Domain in NORMA can have a name. The operations of modification and multiplication are defined on the domains. Domain indexes aren't specially declared, they are introduced in domains' declaration. Domain can be *conditional* and *unconditional*. Conditional domain consists of the index space points. The number and the coordinates of these points can change depending on fulfilment or non fulfilment of *conditions on domain*. Unconditional domain consists of the points from index space, which number and coordinates may be defined during the translation.

There are two different terms in NORMA *declaration of domain* (named conditional or unconditional domain) and *use of domain* (syntactically it is the *name of domain* or *new domain without name*). Domains are used in declarations of the variables declared on domain, setting computation domain in **ASSUME** operators, input and output variables declarations, setting domains of actual parameters in part or functions calls. in reduction functions.

5.1.1.1 *Declaration of unconditional domain.*

declaration-of-rectangular-domain :

multidimensional-domain

new-domain

multidimensional-domain :

one-dimensional-domain

[name-multidimensional-domain :] (domain-product)

domain-product :

component-domain { ;component-domain }+

component-domain :

multidimensional-domain

name-unconditional-domain

onedimensional-domain :

[name-onedimensional-domain :] (name-index=value)

value :

range

const-expression

range :

const-expression .. const-expression

new-domain :

[name-new-domain:] new-domain-without-name

new-domain-without-name :

name-unconditional-domain /list-modification

modification :

name-index=value

name-onedimensional-domain { [+,-] boundary-function}+

boundary-function :

LEFT(*const-expression*)

RIGHT(*const-expression*)

name-rectangular-domain :

name-onedimensional-domain

name-multidimensional-domain

name-new-domain

declaration-of-diagonal-domain :

name-diagonal-domain : name-unconditional-domain / **list**-*condition-on-index*

The notion of one-dimensional domain is a key notion in rectangular dimensions declaration. One-dimensional domain is used for setting the range of points on some coordinate axe in the index space. In the simplest case the name of one-dimensional domain, index name and the boundaries of index values changing are declared in one-dimensional domain declaration.

RegionK: (k =1..15) .

Name-onedimensional-domain can be used for reference to this domain.

The name of index is an index variable. The set of its values is defined by the range const-expression .. const-expression. The boundaries of the range are integer positive constant expressions built from integer constants, domain's parameters and arithmetical operations. Particular values must be assigned to the domain's parameters in the declaration of domain's parameters. The value of the low boundary mustn't be more than the value of upper boundary. For example.

RegionK: (k = 1..15) .

Multidimensional domain is built by operation ";" of rectangular domains multiplication.

It is an example of two-dimensional domain declaration. This domain is built by the multiplication of one-dimensional domains **AxisK** and **AxisL**:

Square: (AxisK: (k=1...15) ;AxisL: (l=1...5)) .

Domain **Square** may be defined in different ways (e.g. taking into account the previous definition):

Square: (RegionK:AxisL: (l=1...5)) .

The ";" operation of rectangular domains' multiplication has an ability :

A;B=B;A if **A** and **B** are domains.

It means that the order of the index space directions in domain's declaration isn't fixed (or fixed arbitrarily, that is the same for user). If the order of index space directions is important (e.g. consistency of directions is needed for using the same variables on the same domains in the different parts) it may be set by **INDEX** declaration (5.1.2).

If the domains are operands of domains' multiplication operation they must have different index names.

The issue of rectangular domain modification may be in addition of some points, deletion of the points or changing the range. The modification of the first two types is defined by the *boundary-functions* **LEFT(n)** and **RIGHT(n).** Function **LEFT** is applied to the left boundary of the range and function **RIGHT** - to the right one. Sign "+" means that the points are added to the one-dimensional domain, sign "-" means that they are deleted. Both functions have one parameter n which defines the number of the points. Integer positive constant may be an actual parameter of the function only. Call to the functions is allowed only in the context with the name of one-dimensional domain setting modificated range:

name-onedimensional domain +**LEFT(n)**

or

name-onedimensional domain+**RIGHT(n).**

E.g. the declaration

FlushK: Square/AxisK-LEFT(2)+RIGHT(2) .

defines domain **FlushK** which consists of points k=3...17.

The boundaries of the resulting range (after using **LEFT** and **RIGHT** functions) must be positive integer and the left boundary mustn't exceed the right one.

Declaration

Square2: Square / AxisK+Left(1)-Right(2) .

isn't right because the left boundary of the resulting range k=0...13 isn't positive.

You can change component one-dimensional domain by the explicit redeclaration of the ranges. For this purpose *index name* of direction and its *new value* must be declared in the modification.

E.g.

Newsquare: Square / AxiisK+RIGHT(3). L=50...80.

If the list of modifications has more than one element modifications declared in that list act on modificated domain in writing order (from the left to the right).

Besides rectangular domain declarations in NORMA there is a possibility of *diagonal* domain settings by the superposition of conditions on previously declared domain (rectangular or diagonal).

Diagonal domain is defined by conditions on indexes of some previously declared domain **D**. It consists of domain **D** points in which the conditions are true. E.g.

KL: ((k=1,...,10) ; (l=1..10)) . Diagonal: KL/ k=1.

set domain **Diagonal** consists of the points (k=1, l=1), (k=2, l=2), ...,(k=10, l=10).

Indexes on variables used in the notation of conditions on indexes can be either internal (from **D** domains set of indexes) or external (5.2.3.2).

The valid form of conditions on indexes defines statically the points of diagonal domain (it is also possible for rectangular domain). That's rectangular and diagonal domain are *static* objects. They differ from *conditional* domain described below.

5.1.1.2 *Declaration of conditional domain.*

declaration-of-conditional-domain :

name--conditional domain , name-conditional-domain :

name-domain / condition-on-domain

Besides static domains in NORMA you can set conditional domain consists of the points from index space. Their number and coordinates can change depending on fulfilment or nonfulfilment *of conditions on domain.*

Here we describe how you can set conditional domain. Previously declared domain **D** is divided into two disjoint subdomains **D1** and **D2**. The first subdomain consists of the points where the conditions on domain are true, the second one - where they are false. Besides **D1** $\cup$ **D2** = **D1**, **D1** $\cap$ **D2** = $\varnothing$.

E.g. set domains:

Domain: ((i=2..n)) ; (j=1..m)) .
Domain1.Domain2: Domain / x+y[i-1, j] - z[j+1] > 0 .

This declaration defines division of initial domain **Domain** into domains *Domain1* and **Domain2** and

Domain1 $\cup$ **Domain2 = Domain,** **Domain1** $\cap$ **Domain2** = $\varnothing$.

Domain1 consists of the points of **Domain** in which the condition $x + y[\, i-1, j\,] - z[\, j+1\,] > 0$ is true. **Domain2** - where the condition is false.

Indexes of the variables used in the conditions notation can be either internal (from **D** domains set of indexes) or external(5.2.3.2).

Program fragment defining conditional domains **Bf2PI** and **BfNot2PI.** and their further division into smaller domains (first domain **BfNot2PI** is divided into two conditional domains **Nodes** and **NotNodes**, and then domain **NotNodes** is divided into two conditional domains **DomainTrue, DomainFalse)** is given below :

DOMAIN PARAMETERS N=3, NV=3 .
Bnu: (Nu=0..2*N).Bf: (II=0..(2*N+1) * (2*NV+1) . BfNu: (Bf:Nu) .
Bf2PI.BfNot2PI: BfNu /II=(2*N+1)*(2*NV+!) AND NU=0.
Nodes.NotNodes: BfNot2PI / ABS (BTNodes -BT) < 0.0001.
DomainTrue.DomainFalse: Nodes / ABS(BTNu - BTNodes) < 0.0001.
VARIABLE BT, BTNodes DEFINED ON Bf DOUBLE .
VARIABLE BTNu DEFINED ON Bnu DOUBLE .

5.1.2 *Declaration of domains' indexes.*

declaration-of-domain-indexes :

INDEX list-*name-index*

The order of the index space directions in domain's declaration isn't fixed (or fixed arbitrarily, that is the same for user). If the order of index space directions is important (e.g. consistency of directions is needed for using the same variables on the same domains in the different parts) it may be set by *declaration of domains' indexes*. The order of index space directions is the same as the order of index's names enumeration in declaration **INDEX** (from the left to the right).

We consider call of part **B** from part **A** with variable **X** values communication from **A** to **B**. Variable **X** is defined on **SquareInA** domain:

```
PART A.
BEGIN
INDEX k,l .
SquareInA.: (AxisK: (k=1...5) ;AxisL: (l=1..10) ).
VARIABLE  X DEFINED ON SquareInA.
  COMPUTE  B( X ON SquareInA).
END PART
PART B. X
BEGIN
INDEX i,j.
SquareInB: (AxisK:  (i=1..5) ; AxisL:  (j=1..10) ) .
VARIABLE  X  DEFINED ON  SquareInB.
END PART
```

Declarations **INDEX** k,l and **INDEX** i,j make correspondence between indexes: **i~k,j~l** . In other words the values of **X** variable are considered in the one and the same way in both parts. If **INDEX i,j** declaration from part B is substituted to **INDEX i,j** the values on corresponding directions will be inconsistent: $5 \neq 10$, $10 \neq 5$.

5.1.3 *Declaration of variables.*

declaration-of-scalar-variables :

VARIABLE list-*name-scalar [type]*

declaration-of-variables-on-domains :

VARIABLE **list**-*definition-variables-on-domain [type]*

definition-variables-on-domain :

list-*name-variable-on-domain*

DEFINED ON *unconditional-domain*

type :

*[***REAL, INTEGER, DOUBLE***]*

Scalar variables and *variables on domain* are *arithmetical variables*. There is a unique name and the type of variable in the declaration corresponding to every arithmetical variable. The types of variables are **REAL, INTEGER** or **DOUBLE** (default **REAL**).

It is an example of scalar 's declaration :

VARIABLE Alpha, X, H. VARIABLE IJK, Z INTEGER.

Every *variable on domain* is connected with the domain in the declaration. This domain defines *index's names*, which can be used in *index expressions* in calls to this variable (the order of index expressions isn't important).There is no special declaration for indexes, they are introduced in domain's declarations.

Square: (AxisK: (k=1..5) ; AxisL: (l=1..5)).

VARIABLE First,Last DEFINED ON Square,

SingleK DEFINED ON AxisK, SingleL DEFINED ON AxisL.

Given declarations define variables **First, Last** on domain **Square.** It means that these variables can have corresponding indexes **k** and **l** in the index expressions. Variables **SingleK** and **SingleL** are defined on domains **AxisK** and **AxisL.** It means that they can have corresponding indexes **k** and **l** in the index expressions. In this call **First[k-1,l+1]** and **First[l+1,k-1]** are equivalent (5.1.2).

In call to *variable on domain* the rule of default indexes setting is performed*: index expressions* which coincide with *index name* may be omitted. E.g. notations **First[k,l], First[k], First[l], First** for variable **First** are equivalent.

If some index is set by constant (constant expression) you must declare which index direction this constant concerns to , e.g. **First[k=5,l-1]**. If you need to link two directions by some formula you must do it in explicit way, e.g. diagonal elements of the matrix **First** may be defined as **First[k,l=k]** or **First[l=k].**

5.1.4. *Declaration of index construction.*

declaration-of-index-construction :

MACRO INDEX *name-index-construction* [**list**-*explicit-index-expression*]

explicit-index-expression :

name-index [{ +,- } *const-expression*]

name-index = const-expression

name-index = name-index [{ +,- } *const-expression*]

Index construction is used for shortening notation of complicated index expressions. It is the simplest case of macrodeclaration. e.g. here are the declarations of index constructions **Short** and **SecondShort** and the declaration of domain **IJKL** and variable **V**.

IJKL :((i=1...10) ; (j=1...10) ; (k=3...17) ; (l=3...18)) .

VARIABLE V DEFINED ON IJKL DOUBLE.

MACRO INDEX Short [i-1, j-1, k+1] .

MACRO INDEX SecondShort [i-1, j-1, k+1, l+2].

This declarations allow notations of **V[Short, l-2]** or **V[l-2,Short]** type for setting variable **V[i-1,l-2, j-1, k+1]** and **V[SecondShort]** for setting variable **V[i-1, l+2, j-1, k+1]**.

5.1.6. *Declaration of domain parameters.*

declaration-of-domain-parameters :

DOMAIN PARAMETERS list-*prescription*

prescription :

name-domain-parameter=int-const

Range boundaries in domain's declaration may be set by parameters of domain (in implicit way). The values of these parameters must be defined in the part. in the domain parameters declaration. E.g.

DOMAIN PARAMETERS M=3, J=90, DomainLeftBoundParameter=15.

Parameters of domain may be included into *arithmetical expressions* and *constant expressions*.

5.1.6 *Declaration of input and output parameters.*

declaration-of-input :

INPUT list-*input-scalar*

INPUT list-*input-on-domain*

input-scalar :

name-scalar [attributes]

input-on-domain :

name-variable-on-domain

declaration-of-output :

OUTPUT list-*output-scalar*

OUTPUT list-*outputs-on-domain*

output-scalar :

name-scalar [attributes]

outputs-on-domain :

list-*output-on-domain [attributes]* **ON** *domain*

output-on-domain :

name-variable-on-domain

attributes :

(list-*attribute* **)**

attribute :

STR(*int-constant* **)**

TAB(*int-constant* **)**

SPACE(*int-constant* **)**

string

ORDER(list-*name-index***)**

LENGTH=*int-constant*

FILE=*'file-name'*

ALL

format

format :

I *int-constant*

{ F.E.D } *int-constant . int-constant*

Declaration of input (output) variables means that all the variables mentioned in the list of input (output) are liable to input (output). E.g. declarations

B1,B2 : B/Z<Eps.

INPUT Velocity ON A. OUTPUT Tau ON B1.

INPUT X, ALPHA.

are requests for input of scalar **X, ALPHA** values , value of **Velocity** in all the points of **B1** domain where condition **Z<Eps** is fulfilled.

The order of variables input (output) isn't set, it is defined during the translation .Minimal piece of information which is input (output) as a whole is *scalar* or *variable on domain. Attributes* may control input (output) of these elements.

Attributes act on the variable (list of variables) declared before. E.g. declaration

INPUT R1.R2 (FILE ='myfile') ON Grid.

is a request for input of variables **R1.R2** values on **Grid** domain (in other words for all the index values of this domain) from file myfile.dat.

Attributes help to control the form of input and output data and bind data with input and output files:

ï **STR(n)** sets the skip of n-1 lines;

ï **TAB(n)** sets *n* spaces before the beginning of the line;

ï **SPACE(n)** sets *n* spaces in the line begging of the current position;

ï *'string'* set output of the textual constant *string*;

ï **ORDER** defines the order of index changing for input; the highest index is the first from the right, the lowest is the first from the left; for one value of every high index the low index has run over all their values ;

ï **LENGTH** sets the length of record in the output file;

ï **FILE** sets the name of input or output files; the method of setting file name is defined during realization (if you use translator version for personal computers input file has a default name with .dat extension; when you set file name this extension may be omitted; if attribute **FILE** isn't given input is carried out from file norma.dat and output to display screen);

ï **ALL** sets output variable's name and its coordinates (index values) for every value of variable on domain.

ï *format* sets format for numerical values for input and output and corresponds to format specificators **I, F, E, D** in the FORTRAN language. There are default formats : E>15.8 for variables of **REAL** type, D15.8 for variables of **DOUBLE** type I5 for variables of **INTEGER** type.

For input variables only attributes of format and FILE type are allowed.

Here is an example of input variables declaration :

```
INPUT One (FILE='file1',F10.2) ON Grid1,
      Two(FILE='file2' ) ON Grid2.
```

Here are examples of output variables declarations:

```
OUTPUT Velocity(STR(5) ,TAB(7), 'Velocity = ' ,F9.1) .
OUTPUT X(FILE='FILE17' , ORDER(J,K,I), ALL, f5.1) ON Grid.
OUTPUT Y('Matrix',SPACE(10),'of values',SPACE(10),'Y parameter',
                FILE='OT5' ,ALL,F15.2) ON ABC.
```

For output in more complicated form (diagrams, tables, etc.) you should use standard libraries and packets tools or your own programs written in other languages.(5.2.5).

Syntax of input data files:

file :

*input-element {input-element }**

input-element :

name-scalar =arithm-constant;

name-variable-on-domain(**list**-*index-range*)=*data;*

index-range :

name-index = int-constant .. int-constant

data :

list-*data-element*

data--element :

arithm-constant

int-constant(data)

Programmer can:

√ place input elements in any order,

√ control the order of numerical values in file by changing index order,

√ not keep to one format in numerical values notation,

√ write repeated data in a short form.

Here is an example of input data file's contents:

```
C(K=1..10)=5(-10.1) ,5(1.01);
ALPHA=3.17; BETA=-0.12; GAMMA=1.0E-10
C(K=11..20)=5(10.1,5(-1.01);
```

Here is an example of input data file data.dat contents for Gauss program:

```
MAIN PART Gauss.
Solution of linear equations by Gauss-Jourdan method.
BEGIN
Ot:(t=0..n). Oi:(i=1..n). Oj:(j=1..n).
Oij:(Oi:Oj). Otij:(Ot:Oij).
Oti:(Ot:Oi). Otij1:Otij / t=1..n. Oti1:Oti / t=1..n.
   DOMAIN PARAMETERS n=5.
VARIABLE a ON Oij. VARIABLE m ON Otij.
VARIABLE b, ON Oi. VARIABLE r ON Oti.
   INPUT a(FILE='data') ON Oij, b(FILE='data') ON Oi.
   OUTPUT x(FILE='results',ALL) ON Oi.
FOR Otij/t=0 ASSUME m=a.
FOR Oti / t=0 ASSUME r=b.
OtiEQtij1,OtiNEtj1:Otij1 / i=t. OiEQti1,OiNEti1:Oti1 / i=t.
FOR OtiEQtj1 ASSUME m = m[t-1,i=t]/m[t-1,i=t,j=t].
FOR OiEQj1 ASSUME r = r[t-1,i=t]/m[t-1,i=t,j=t].
FOR OiNEtj1 ASSUME m = m[t-1]-m[t-1,j=t],m[i=t].
FOR OiNEti1 ASSUME r = r[t-1]-m[t-1,j=t],m[i=t].
FOR Oi ASSUME x = r [t=n].
END PART.
```

Input file data.dat:

```
B(I=1..3)=5.0, 13.0, 3.0;
A(I=1..5,J=1..5)=2.0, 3.0, -4.0, 5.0, -1.0,
 3.0, 4.0, -1.0, 6.0, 1.0,
```

2.0, 0.0, -3.0, 0.0, 4.0,
0.0, 2.0, 0.0, 0.3, 0.0,
3.0, -1.0, 2(0.0), 1.0;
B(I=4..5)= 5.0, 3.0;

The values of data multidimensional arrays are placed in the file corresponding to the given indexes. The values of the first index from the right are changed the first : in the given example matrix A(I,J) is set on rows, in other words I=1, J = 1,2,3.4,5, then I = 2, J = 1,2,3,4,5 etc. till I=5, J = 1,2.3,4,5.

The way of input file data.dat setting isn't the only one.

5.1.7 *Declaration of external names.*

declaration-of-external-names :

declaration-of-external-functions

declaration-of-external-parts

declaration-of-external-functions :

EXTERNAL FUNCTION list-*name-external-function [type*]

declaration-of-external-parts :

EXTERNAL PART list-*name-external-simple-part*

If the names of the functions or parts are their actual or formal parameters they are indicated In the *declaration of external names* by all means . Default type of external function is **REAL.**

EXTERNAL FUNCTION Last,First DOUBLE.

EXTERNAL PART Middle.

5.1.8. *Declaration of distribution indexes.*

declaration-of-distribution-indexes :

DISTRIBUTION INDEX *name-index = simple-range [, name-index = simple-range]*

simple-range :

int-constant[.. int-constant]

Declaration of distribution indexes is used for representation of two index directions from task domain index space on processor element (PE) matrix of distributed system. If functions and parts have this declaration they are called *distributed*. if there is no such declaration in their content they are called *nondistributed* systems. You can meet this declaration in the distributed part or function only once; it is prohibited in the main part in other words the main part is always nondistributed.

Given declaration results in data and control distribution between PE elements of the system or automatic generation of data exchange between PE if you need it. The subject of distribution is variables which indexes coincided with ones in declaration **DISTRIBUTION INDEX**. E.g. if there is a declaration

DISTRIBUTION INDEX i=2..8, j=11.

all the variables defined on the domains with i and j indexes will be distributed on $PE_{i,j}$ with virtual row numbers i=2..8 in the column number j=11 of PE matrix.

Computations defined in nondistributed part or function are carried out as a whole in one PE (though there may be several such PEs).

Declaration

DISTRIBUTION INDEX i=2...8,j=0.

is false: the elements of PE matrix are considered to be numbered from 1.

5.2 Operators in NORMA.

operator :

scalar-operator

operator-**ASSUME**

call-part

There are three defined types of operators in NORMA. They are scalar operator, operator **ASSUME** and call part. The operators are used for description of computations for task solution.

5.2.1 *Scalar operator.*

scalar-operator :

name-scalar = scalar-arithm-expression

scalar-arithm-expression :

arithm-expression

Scalar operator is used for computation of scalar arithmetical values . In fact it is an analogue of assignment statement in traditional programming languages. Name of scalar is indicated in the left part of the operator and scalar arithmetical expression built in usual way from scalars, arithmetical constants, domain's parameters, function calls, variables on domain with index-constants.

Variables defined on domain which index expressions are not constants can't be included into scalar arithmetical expression (exception from this rule is arguments of reduction functions (5.2.3)).

IJ: ((i=1..MaxI) ; (j=1..MaxJ)).

DOMAIN PARAMETERS MaxI = 3, MaxJ = 90.

VARIABLE ScalarV INTEGER. VARIABLE Pi DOUBLE.

VARIABLE ArrayV DEFINED ON IJ DOUBLE.

ScalarV = MaxI.(MaxJ-1)/2+SQRT(PI)/SIN(ArrayV[i=1,j=55]).

Reassignment in scalar operator is prohibited, that's the next operator is false:

ALPHA = ALPHA-1.

5.2.2 **ASSUME** *operator.*

operator-**ASSUME**

FOR *domain* **ASSUME** *relation{;relation}**

relation :

name-variable-on-domain = arithm-expression

call-part

ASSUME operator is used for computation of arithmetical variables defined on domains.

The case of *part call* in the body of **ASSUME** operator is described in 5.2.4.

The terms of arithmetical expression in the right part of relation may be variables defined on domain, scalars, arithmetical constants, domain's parameters, function calls, indexes.

Informally **ASSUME** operator's semantics is defined in the way described below. Let's consider given relation

FOR D$(i_1,...,i_n)$ **ASSUME**

$$X^1_{IndL(i_1,...,i_n)} = F(X^{j_1}_{IndR^{j_1}(i_1,...,i_n)},...,X^{j_k}_{IndR^{j_k}(i_1,...,i_n)},Other)$$

where

- **D**$(i_1,...,i_n)$ - **ASSUME** operator's domain.
- $(i_1,...,i_n)$ - **D** domain indexes,
- X^{j_q}, $1 \le q \le k$ - variables names defined on domain,
- $IndL(i_1,...,i_p)$, $1 \le p \le n$ -index expressions of the left part
- $IndR^{j_q}(i_1,...,i_n)$ - index expressions of the right part,
- F - function computed in the right part,
- $Other$ - other terms of the right part.

Every relation set the rule F of variable X^1 from the left part values computation based on the values of X^{j_1} ..... X^{j_k} variables and the terms $Other$ from the right part:

(1) all the points $(a_1,....a_n) \in \mathbf{D}(i_1,.....i_n)$ are defined;

(2) value of X^1 from the left part is computed in $IndL(a_1,.....a_n)$ point for every point $(a_1,....,a_n) \in \mathbf{D}$;

(3) index expressions $IndR^{j_q}(a_1,...,a_n)$ of all the variables X^{j_q}, $1 \le q \le k$ from the right part of relation are computed for every point $(a_1,...,a_n) \in \mathbf{D}$ and the set of right part arguments is defined

$$\mathbf{X}(a_1,...,a_n) = \{X^{j_q}_{IndR^{j_q}(a_1,...,a_n)}, Other\}, \quad 1 \le q \le k$$

(4) if for some period of time for point $(a_1,...,a_n) \in \mathbf{D}$ all the arguments from $\mathbf{X}$ have been computed the computation of $X^1_{IndL(a_1,...,a_n)}$ value from the left part is possible, but if all the arguments haven't been defined the computation in that point in that period of time is impossible (it doesn't mean that it is impossible in all cases).

The name of **ASSUME** operator underlines the fact that it defines the rule of value computation in the only way but doesn't require immediate computation in particular place of program and doesn't arrange the order and the method (parallel or sequential) of computation.

E.g. if the variable is defined as

Matrix: ((I=1..5) ; (j=1..5)).

VARIABLE DEFINED ON Matrix.

then operator

FOR Matrix ASSUME X =0.

is a request for 25 elements of variable **X** zeroisement. The method of this request realization isn't fixed in this language.

NORMA is a single-assignment language, reassignment is prohibited. That's further operators are incorrect:

FOR Matrix ASSUME X = Y; X = Z.

FOR Matrix ASSUME X = X+1.

Constraints on the index form the left part of relation (these are indexes without displacement) are not essential because domain **D** from the operator's header allows to define rather complicated relations. If you need to define the computation of

$X_{i,j} = F(Y_i, Zj)$ on domain **Ox : ((I=1..N) ; (J=1..M)).**

type you may do it in this way :

Ox.JI : Ox/ J=I.

FOR Ox.JI ASSUME X = F(Y,Z) .

As we mentioned earlier functions may be included into arithmetical expression and scalar arithmetical expression.

5.2.3 *Function calls in Norma.*

call-function :

call-reduction-function

call-standard-function

call-external-function

call-reduction-function :

name-reduction-function

((name-domain)arithm-expression)

call-standard-function :

name-standard-function (arithm-expression)

name-reduction-function :

{ SUM,MULT,MAX,MIN }

call-external-function :

*name-external-function[(**list**-in-parameter)]*

in-parameter :

arithm-expression

name-variable-on-domain **ON** *domain-of -parameter*

iterated-variable-on-domain ***ON*** *domain-of-parameter*

name-external-simple-part

name-external-function

domain-of-parameter :

name-unconditional-domain

name-unconditional-domain / index-expression

*name-unconditional-domain / (**list**-index-expression)*

iterated-variable-on-domain :

name-variable-on-domain [name-iteration-index [-1]]

There are defined *standard arithmetical functions, reduction functions* and *external user's functions* in the language.

5.2.3.1 *Standard arithmetical functions.*

Functions : *sqrt(x), abs(x), esp(x), alog(x), alog10(x), sin(x), cos(x), asin(x), atan(x), entier(x)* are standard functions.

5.2.3.2 *Reduction functions*

Functions : **SUM** (sum), **MULT**(multiplication), **MAX**(maximum), **MIN**(minimum) are reduction functions. Call to these functions :

name-function((name-domain) arithm-expression).

Domain defines the number of domain's point where the function will be computed, arithmetical expression defines the number of values which the function are applied to. Here is an example :

$$V_i = W_i + \sum_{j=1}^{m} A_{i,j} * X_j, \quad i = 1,...,n.$$

The fragment of this computation in NORMA :

```
Grid: (Oi: (I1..N) ; Oj: (J=1..M) ).
VARIABLE A DEFINED ON Grid.
VARIABLE V,W DEFINED ON Oi.  VARIABLE X DEFINED ON Oj.
FOR Oi ASSUME V = W + SUM( ( OJ ) A*X) .
```

Here is an example of embeded reduction functions:

```
SurfaceMax:
    (Left: (Three:(IV=1..3) ; A:(J=0..2*M);B:(I=0..2*N) );
    Right: ( ( IK=1..9) ; (W=0..2*M) ; BNU:(NU=0..2*N) ) ).
SUMPOL:( (W=0..2*M) ;BNU).
ThreeAB:(Three;A;B).  OLINE:(LINE=1..3).
VARIABLE VV,V DEFINED ON ThreeAB DOUBLE.
VARIABLE JSEQ DEFINED ON SurfaceMax DOUBLE.
FOR ThreeAB ASSUME
  VV=SUM( (SUMPOL)
     SUM((OLINE) V[IV=LINE] *
          (V[IV=1] * JSEQ[IK=LINE] + V[IV=2] * JSEQ[IK=LINE+3] + V[IV=3] * JSEQ[IK=LINE+6]) ) ) .
```

The rule of index localization is true for reduction functions : domain declared as argument of reduction function has its own index system. The subject of this system action is arithmetical expression declared as the second argument of reduction function. It means that indexes used in the arithmetical expression may be divided into two types : "internal" which coincide with the indexes of the reduction domain and defined by them and "external" which don't coincide with indexes of reduction domain and defined by external domains (e.g. domains of other reduction functions or **ASSUME** operator). The set of "internal" indexes values is completely defined by the domain set as an argument of reduction function. The set of "external" indexes values is defined by the domains which are external for reduction function.

E.g. operator

```
FOR Oik ASSUME X=(i+k) / SUM( (Oij5) B.C.
```

defines request for computation

$$X_{i,k} = \frac{i+k}{\sum_{i,j=1}^{5} (B_{i,j,k} * C_{i,j})} \quad i,k = 1,....10$$

if the declarations are :

```
Oij55:  ((i=1..5) ; (j=1..5)). Oik:  (i=1..10) ; (k=1..10)).
Oijk:   (Oij:((i=1..15) ; (j=1..15)) ;(k=1..15)).
```

VARIABLE B DEFINED ON Oijk. VARIABLE C DEFINED ON Oij.

VARIABLE X DEFINED ON Oik.

If external domain is conditional the condition acts on all the external indexes of arithmetical expression which are under the sign of reduction function. E.g. formula

$$X_{i,j} = \sum_{i=1}^{100} B_{i,j} Y_j \quad i = 1,...,10, \quad j = 1,...,10, \quad i < j, \quad Y_i < 0,$$

sets computation

$$Y_2 < 0: \quad X_{1,2} = \sum_{i=1}^{100} B_{i,2} Y_2 \quad ... \quad Y_{10} < 0: \quad X_{1,10} = \sum_{i=1}^{100} B_{i,10} Y_{10}$$

$$Y_3 < 0: \quad X_{2,3} = \sum_{i=1}^{100} B_{i,3} Y_3 \quad ... \quad Y_{10} < 0: \quad X_{2,10} = \sum_{i=1}^{100} B_{i,10} Y_{10}$$

.....

$$Y_{10} < 0: \quad X_{9,10} = \sum_{i=1}^{100} B_{i,10} Y_{10}$$

may be written as :

Oij: (Oi: (i=1..10); Oj: (1..10)) . OiLTj:Oij / i<1.
OiLTjYLTO, OiLTjYGEO:OiLTj / Y<0.
Oi100: (i=1..100). Oij100: (Oi100;Oj).

VARIABLE B DEFINED ON Oij100. VARIABLE X DEFINED ON Oij.

VARIABLE Y DEFINED ON Oj.

FOR OiLTjYLTO ASSUME X=SUM((Oi100) B*Y).

Recursive computations are prohibited for the values used in both the left part of **ASSUME** operator and arithmetical expression in reduction function. E.g. formula

$$X_{i,j} = \sum_{\substack{i=1 \\ i<j}}^{10} X_{i,j} \quad i = 2,...,10, \quad j = 1,...,5$$

5.2.3.3 User's external functions.

External function is defined by user. One can call this function by name, the list of actual parameters divided by comma is put into brackets and placed after the name. The result of function's computations is represented by function's name. If all the actual parameters have their values computation is available . All the actual parameters are considered initial data for function computation, side effect is prohibited.

Arithmetical expressions, part and function names or variables on domain (arrays) can be actual parameters of external function.

Here is an example of external function with different ways of actual parameters setting .

Grid: (Oi: (I=1..N) ; Oj: (J=1..M)).

VARIABLE A, B DEFINED ON Grid.

VARIABLE X,Y DEFINED ON Oi. VARIABLE T DEFINED ON Oj.

VARIABLE Gamma DOUBLE.

INPUT A,B ON Grid, Y ON Oi, T ON Oj. INPUT Gamma.

OUTPUT X ON Oi.

DOMAIN PARAMETERS N = 10, M = 20. EXTERNAL FUNCTION DxDy.

FOR Oi ASSUME X = F(DxDy, Gamma+0.5,Y, T ON Oj, A ON Grid / (i=2..8,J=M-2), B ON Grid /J=I+2).

ASSUME operator set the rule of array $X_{i,i}$ $=1,10$ computation. User's function F with parameters values are to be computed for every i. The list of parameters :

1-st parameter - name $DxDy$ of external user's function;

2-nd parameter- arithmetical expression $Gamma+0.5*Y_i$;

3-rd parameter - $T_{j,j}$ $=1,...,20$ values of static array;

4-th parameter - $A_{i,j}$, $i=2,...,8$, $j=18$ values of array's section : one-dimensional array consists of the 2-8 rows elements from the 18th column of **A** matrix;

5-th parameter - $B_{i,j}$, $i=1,....,10$, $j=i+2$ of dynamic array's section : i+2 column of **B** matrix.

The rules of formal and actual parameters correspondence :

(1) The number of actual and formal parameters must be equal; correspondence is set from the left to the right in writing order.

(2)

Formal parameter	Actual parameter
function name	function name
part name	part name
scalar	arithmetical expression
variable on domain	variable on domain

(3) If both formal and actual parameters are variables on domain then the number of domain indexes and the number of points in the ranges on corresponding indexes must be the same.(5.1.2).

5.2.4 *Part call.*

call-part :

COMPUTE *name-part [(actual-parameters)]*

actual-parameters :

*[***list***-in-parameter]*

*[***RESULT list***-out-parameter]*

out-parameter :

name-scalar

name-variable-on-domain ON domain-of-parameter

iterated-variable-on-domain ON domain-of-parameter

Arithmetical expressions, part and function names or variables on domains (arrays) can be actual parameters.

Actual parameters may be declared as initial or results by key word **RESULT**.

E.g.:

COMPUTE Velocity (Delta+0.5, Fi ON Oijk RESULT V ON Oij).

The first two parameters are initial , the third is result. Side effect is impossible : if sets of initial parameters and parameters-results intersect the result of intersection will be reassignment which is prohibited in NORMA.

Part call is a development of notion relation in **ASSUME** operator because it allows to obtain several results at once. Part call without **ASSUME** operator is a development of scalar operator notion.

The ways of part initial actual parameters setting and the ways of function actual parameters settings (5.2.3.3) are the same.

Scalars, variables with indexes(may be set by the rule of default indexes), variables on domains (arrays) may be actual parameters-results.

If part call is in the body of **ASSUME** operator then scalars and variables on static domain (which don't change for different values of indexes from **ASSUME** header) can't be parameters-values - it causes reassignment.

If part call isn't in the body of **ASSUME** operator then scalar names, variables with index-constants and variables on static domains can be parameters-results only.

Here is an example of actual parameters settings in different ways .

VARIABLE A,B DEFINED ON Grid: (Oi: (I=1..N);Oj: (J=1..M)).

VARIABLE X,Y DEFINED ON Oi. VARIABLE T DEFINED ON Oj.

VARIABLE Gamma DOUBLE.

INPUT A ON Grid ,Y ON Oi,T ON OJ. INPUT Gamma.

OUTPUT X ON Oi, B ON Grid /J=2..12.

DOMAIN PARAMETERS N =10. =20.

EXTERNAL PART DzDy.

FOR Oi ASSUME

COMPUTE TEST (DzDy, T ON Oj, Y RESULT X, B ON Grid /J=I+2).

COMPUTE SCALAR(A ON Grid RESULT Gamma, B ON Grid /J=2).

The rule of $X_{i,j,}$ $i=1,...,10$ and part of matrix **B** : $B_{i,j,}$ $i=,...,10,$ $j=3,...,12$ computation is defined in **ASSUME** operator.

Scalar call of part **SCALAR** defines scalar **Gamma** and the second column of **B** matrix.

The rules of actual and formal parameters-results correspondence are the same as for correspondence rules for functions(5.2.3):

Formal parameter	Actual parameter
scalar	scalar
scalar	variable with index constant
variable on domain	variable on domain

5.2.5 *Interface with FORTRAN programs.*

Subroutines and functions written in Fortran may be called from the program written in NORMA by usual tools of part and function call. Reassignment control is carried out in actual parameters analysis.

For example, subroutine

```
SUBROUTINE  SNIPPY(Y,N,X)
REAL  X(N),  Y(N)
DO  1  I=1,N
X(I)  =  SIN(Y(I))
1  CONTINUE
RETURN
END
```

can be called as it is shown here:

```
Oi  : ( I=1..N ).
VARIABLE  X,Y  DEFINED ON  Oi.
INPUT  Y ON  Oi.
DOMAIN  PARAMETERS  N = 10.
   COMPUTE  SINXY(Y ON Oi, N  RESULT X ON Oi).
```

5.2.6 Setting of sequential computing mode.

The order of carrying out operations (operators) is often important in computational algorithms realization. The order may have an influence on convergence, stability of solution method etc. There is a possibility of sequential computations mode setting in Norma. It allows user to fix the order of computation. On this purpose delimiters # ... # are used. Operators put in the delimiters # ..# are performed in the order written in program. Correctness of operators' sequence in Norma is under control : reassignment, using of undeclared values , etc. are fixed. The notation

#

X=5.0 . Z=SIN(X+0.5) . Y=COS(X)-Z.X.**

#

is correct. but notation

#

X=5.0. Y=COS(X)-Z.X . Z=SIN(X+0.5) .**

#

is incorrect because undefined value of Z variable is used in the second operator(if we take off delimiters then both notation are correct).

5.3 Iteration

iteration :

head-of-iteration

[boundary-value]

initial-value

body-of-iteration

exit-condition

end-iteration

head-of-iteration :

ITERATION list-*iterated-element*

ON *name-iteration-index* .

iterated-element :

name-variable[(**list**-*name-result*)]

boundary-value :

BOUNDARY [*operator.*]$^{+}$ **END BOUNDARY**

initial-value :

[**INITIAL** *name-iteration-index* =

int-constant:{element-of-initial.}$^+$

END INITIAL*]*$^+$

element-of--initial :

operator

declaration-of-input

declaration-of-output

body-of-iteration :

{element-of-iteration-body.}$^+$

element-of-iteration-body :

operator

iteration

declaration-of-output

exit-condition :

EXIT WHEN *log-expression*

end-iteration :

END ITERATION *name-iteration-index*

Computation process often is iterative in mathematical physics problems solutions. Such process may be set by previously described tools of the NORMA language. You should extend domains' declarations adding extra direction corresponding to iteration index. But to use such an extension isn't always correct because the direction is fictitious and it represents the way of computation but not space-time grid. Besides the boundaries of such fictitious direction are often unknown.

Special construction **ITERATION** allows possibility to set iterative computational process avoiding all the difficulties mentioned above.

Informally iteration set iterative computations with iteration index changing from 0 to some integer positive value which is a condition of iteration's end.

Let's take iterative computational process dealt with the system of equations solution:

$$\sum_{j=1}^{m} A_{i,j} X_j = F_i, \quad i = 1,\ldots,m$$

and set by formulae

$$X_i^{n+1} = \frac{1}{A_{i,i}}(F_i - \sum_{i \neq j} A_{i,j} X_j^n). \quad X_i^0 = X0. \quad i = 1,\ldots,m$$

exit condition $\left\| X^n - X^{n-1} \right\| < \varepsilon$

This process written in NORMA:

```
Array: (Oi: (I=1..M) ;Oj: (J=1..M)) .
VARIABLE X0,X,F DEFINED ON Oi. VARIABLE A DEFINED ON Array.
VARIABLE Epsilon.
DOMAIN PARAMETERS M = 100.
INPUT X0 ON Oi, A ON Array. INPUT Epsilon.
OUTPUT X, Xpred ON Oi.
ITERATION X (Xpred) ON N.
   INITIAL N=0 :
   FOR Oi ASSUME X = X0.
   END INITIAL
FOR Oi ASSUME
  X = 1/A[j=1].(F-SUM( (Oj/i<>j)A.X[I=J,N-1]).
EXIT WHEN MAX( (Oi) ABS(X[N] - X[N-1] ) ) < Epsilon
END ITERATION N.
```

Iteration itself is the last 8 lines of example. There is iteration index (here it is **N**) and variables taking part in computations and presenting the result of these computations (- **X** -is the value from the last iteration step, - and **Xpred** — the value from the before the last iteration step) in the iteration header. Value **Xpred** doesn't require supplementary definition, it is considered to be declared in the same way as **X**. Values **X** and **Xpred** can be used for computations out of iteration: in the given example these values are declared output.

You can use the possibility of iterated variables boundary values of setting (in general it is not obliged). Boundary values of iterated variable are set by common Norma operators inside the block **BOUNDARY ... END BOUNDARY**. These values are considered unchangeable on iteration and defined on every its step.

Initial values for iterated variables are set by blocks **INITIAL...END INITIAL** (there may be several ones).If the values of iterated variables depend only on the values of the previous level (one-level iteration) then only one block is indicated

INITIAL N=0:... END INITIAL.

If the values of iterated variable depend on the values of two previous levels (two-level iteration) then two blocks are indicated

INITIAL N=0: ... END INITIAL

INITIAL N=-1: ... END INITIAL

etc.

Variables which are the result of multilevel iteration and used outside iteration are indicated in the header of iteration. E.g. the header

ITERATION X(X1,X2,X3) ON N.

defines values **X,X1,X2,X3** of iterated value **X** obtained at the last four steps of four level iteration. The number of results can't be more than *n* for *n*-level iteration.

Body of iteration is part of NORMA program. In particular you can define new iteration in another direction inside the iteration. In the given example body of iteration consists of the only **ASSUME** operator.

Iteration index can be used in the list of indexes indicated for iterated variable. In fact we can consider that iterated variable has supplementary index in boundaries of iteration (on factitious direction). Iteration index without displacement may be not indicated.

Only iterated variables can indicate iteration index.

The values of variables unindicated in the list of iterated variables may be computed in the body of ITERATION construction. Such variables may be used for interim results of each level iteration representation. The usage of such variables doesn't break reassignment prohibition : we consider the a new copy of the variable is used at each step of iteration.

Iteration process ends if logical expression set in exit condition becomes true.

Appendix 1. Representation of initial program.

Initial program is represented in initial file according to the rules :

1. Text of every part of NORMA program is written in unformatted representation. If you need to carry operator or declaration it is prohibited to break key words, identificators, constants. In other cases you can carry the words as you like it.

2. Key words, identificators, constants are separated by *spaces, special symbols, end-line symbol.*

Spaces are not meaningful symbols: the group of spaces are considered as one space.

3. The line or part of line beginning from symbol "!" is a *commentary.*

4. Information placed in the interval from the symbol "?" standing at the first position up to the symbol "?" also standing at the first position are not translated. It allows to choose from initial file those parts which you want to translate.

Bibliography

1. **Zadykhailo I.B.** Organization of loop computations based on parameter record of special type. *Computational mathematics and mathematical physics journal*, **3**, N 2, 1963, p.337- 357 (in Russian).
2. **Andrianov A.N., Efimkin K.N., Zadykhailo I.B., Podderugina N.V.** The NORMA language. *Preprint KIAM AS USSR*, 1985, N165, 34p. (in Russian).
3. **Zadykhailo I.B., Pimenov S.P.** Algorithms of output program synthesis on NORMA specifications. *Preprint KIAM AS USSR*, 1986, N 141, 24p. (in Russian).
4. **Zadykhailo I.B., Pimenov S.P.** The NORMA language semantics. *Preprint KIAM A S USSR*,1986, N 139, 22p. (in Russian).
5. **Pimenov S.P.** Semantics of the NORMA language and algorithms of output program synthesis. *Ph.D. Thesis*, Moscow, 1986, 12p. (in Russian).
6. **Andrianov A.N.** The Synthesis of loop computations in the NORMA language. *Preprint KIAM A S USSR*, 1986, N 171, 28p. (in Russian).
7. **Andrianov A.N., Efimkin K.N.** Some questions of program synthesis the class of mathematical physics problems. *Thesis of IV All-Union conference "Applications of mathematical logic methods"*, Tallinn, 1986, p.22-23 (in Russian).
8. **Andrianov A.N., Efimkin K.N., Zadykhailo I.B.** Nonprocedural NORMA language and methods of its implementation. *Algorithms and algorithmic languages*. Moscow, Nauka, 1990, p.3-37. (in Russian).
9. **Andrianov A.N.** The synthesis of parallel and vector programs on nonprocedural NORMA specification. *Ph.D.Thesis*, Moscow, 1990, 144 p. (in Russian).
10. **Andrianov A.N., Efimkin K.N., Zadykhailo I.B.** Nonprocedural language for mathematical physics problems solution. *Programmirovanie*, N2, 1991, p.80-94. (in Russian).
11. **Andrianov A.N., Efimkin K.N., Zadykhailo I.B.** Nonprocedural NORMA language and methods of its implementation. *Thes. First Int. Workshop "Software for Multiprocessors and Supercomputers: Theory, Practice, Experience"*, St.Petersburg, Russia. Febr.14-20,1993.
12. Andrianov A.N., Efimkin K.N., Zadykhailo I.B . Nonprocedural language for Mathematical Physics. *Programming and computer Software*, 17, 1992.N2, p.10-22.
13. Bugerya A.B. Implementation of the NORMA language's mathematical functions for distributed computational systems. *Preprint KIAM RAS* ,1994, N 37, 26p. (in Russian).

14. **Andrianov A.N., Efimkin K.N., Zadykhailo I.B.,** Nonprocedural language NORMA and Problems of its Implementation. *Proceedings SMS TPE'94*. Moscow, Russia Sept. 19-23, 1994, p.523-530.

15. **Andrianov A.N., Vasiliev M.M., Efimkin K.N., Zadykhailo I.B., Ivanova V.N.** On solving the three-dimensional stationary viscous incompressible fluid flow problem by the method of potentials. *Preprint KIAM RAS* ,1992, N 62, 27 p. (in Russian).

16. **Vasiliev M.M., Efimkin K.N., Ivanova V.N.** On the application of the hydrodynamic potentials method to the viscous fluid flow problem. *Matematicheskoe modelirovanie,* 1992, v.6, N 10, p .58-65. (in Russian).

17. **Lubimsky E.Z.** Problems of programming automation. *Bulletin of AS USSR*, 1960, N8.

Keldysh Institute of Applied Mathematics
Academy of Science USSR

Andrianov A.N.

Organization of loop computations in the NORMA language.

Moscow 1986

"Organization of loop computations in the NORMA language".

A.N. Andrianov,KIAM AS USSR

Preprint n 171, M.,,1986,p.28, bibl.4

The problems of loop process organization for the program written in nonprocedural language NORMA are considered in this paper. An algorithm of designing the system of simple loops allowing parallel processing is given. The algorithm is based on the notion of computation's front which is a hyperplane where variables' values may be computed in every its point. The task of Linear Integer Programming is solved for determination of the hyperplane's parameters.

Key words: nonprocedural language, synthesis of the program, parallel computations.

Contents.

1. Introduction.

The NORMA language is used for writing numerical solutions of mathematical physics problems. Program in NORMA is a strict notation of formulae and some supplementary information for a translator to design computer program.

Program has no information about the order of computation and the ways of loop process organization. Determination of computation order is one of the main translator's tasks. Information connections set in the program are initial data for computation order's determination and organization of loop process.

Nonprocedural nature of NORMA allows to overcome some difficulties in algorithm parallelization.

An algorithm of loop process organization for NORMA program is given in this paper. Loop parallel processing can be organised by this algorithm.

Principal NORMA operator.

There are declarative and executive parts in NORMA program. Variables ' names types which are to calculate in the task's computation are indicated in declarations. The rules of carrying out computations according to the language semantics are placed in the executive part [2].

Let's describe briefly principal language operator ASSUME. The structure of the operator is given below:

FOR <domain > ASSUME <relation>.

Let's determine informally the terms "domain" and "relation" . The domain is a set of points with integer non negative coordinates in n-dimensional space.

The form and the type of relation is given below:

$$X^{l}_{i_1,i_2,\ldots,i_n} = f_l(X^{1}_{A},\ldots,X^{m}_{A_m}), \qquad l=\overline{1,m}$$

Here X^{l} are variables names. Set $\{i_1, i_2,\ldots,i_n\}$ is a set of indexes. Index construction A_k $k=\overline{1,m}$ has the following structure:

$$i_{k_1} \pm \Delta_{k_1},\ldots,i_{k_\mu} \pm \Delta_{k_\mu} \quad 1 \le \mu \le n$$

Δ_{k_j} - integer non-negative constants.

Relation sets the rule of deriving X^{l} value from the left part of the relation by the values of variables from the right part. Any variable is calculated only once in every point of domain i.e. reassignement is prohibited in NORMA. Operator ASSUME points out the necessity of computing the variable's values from the left part of the relation for all the points of the domain set in the operator's

header. In general it doesn't mean that computing of X' will be carrying out simultaneously for all the points of the domain in the particular place of the program.

Example. Let two operators be set:

1. FOR A: (k=1..m) ASSUME
   $X=f_1(Y_{k-1})$.
2. FOR A ASSUME
   $Y=f_2(X_k)$

We consider value of variable Y for K=0 has been calculated before these operators. It is evident that operators order where all the values of X variable are calculated first and Y values are done after them is impossible. Argument is to be computed for calculating X even when K=2 (here argument is Y value for K=1). Y is calculated in the second operator i.e. X and Y components are computed by turn in the given case.

Determination of the operators' order.

One of the problems to be solved on the translation stage is the problem of operators (or the groups of the operators) order determination.

To specify a program as a dependence graph is a traditional way of program representation and we use well-known approaches and methods in our work.

Let's determine dependence graph $G(V, E, \lambda)$ which is a directed graph. V denotes a set of nodes corresponding to the names taking part in the computations. If variable X is calculated in several operators it corresponds to several nodes. E denotes a set of arcs corresponding to relations between variables. Arc $(\Delta_{k_1},\ldots,\Delta_{k_n})$ from the node corresponding to variable X'_1 to the node corresponding to variable X' conforms to the dependence X' on the values of X'_1 with index structure $i_{k_1} \pm \Delta_{k_1},\ldots,i_{k_n} \pm \Delta_{k_n}$. λ denotes a set of arcs' marks.

Strongly connected subgraph of graph G not being included into any other strongly connected subgraph of graph G is called the Most Strongly Connected Subgraph (MSCG) of graph G. Having chosen all the MSCG $G_1,\ldots,G_n$ (e.g. by the method suggested in [3]) partial arrangement of $G_1,\ldots,G_n$ could be done: if there is an arc from G_i to G_j, then $i > j$.

Consider loop program realization $R(G_i)$ built for every G_i. Then the program on the set graph G can be represented in the form of realizations' $R(G_i)$ $i=\overline{1,n}$ sequence taking into

consideration partial arrangement set earlier. Thus let's consider an algorithm of designing loop realization for the operators corresponding to the nodes of MSCG.

Loop program- realization is designed for particular operators. Further we'll determine the class of these operators.

Consider relation system for computing values $\overset{1}{X},\ldots,\overset{m}{X}$:

$$\overset{1}{X}_{i_1,i_2,\ldots,i_n} = f_1(\overset{1}{X}_{i_1\pm\Delta^1_{11},\ldots,i_n\pm\Delta^1_{1n}},\ldots,\overset{m}{X}_{i_1\pm\Delta^m_{11},\ldots,i_n\pm\Delta^m_{1n}})$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (1)$$

$$\overset{m}{X}_{i_1,i_2,\ldots,i_n} = f_m(\overset{1}{X}_{i_1\pm\Delta^1_{m1},\ldots,i_n\pm\Delta^1_{mn}},\ldots,\overset{m}{X}_{i_1\pm\Delta^m_{m1},\ldots,i_n\pm\Delta^m_{mn}})$$

Initial domain is set by inequalities:

$$M_k \le i_k \le N_k \quad k=\overline{1,n} \qquad (2)$$

System (1) is said to correspond to some MSCG.

Task may be formulated in the following way:

Design loop program-realization which provides computation of variables $\overset{1}{X},\ldots,\overset{m}{X}$ for all the points of the domain (2) if such computation is possible. Program realization must be designed according to the given system of relations (1).

Front of computations.

Further we'll consider computation organization only in the formulae corresponding to MSCG. Considered model of computations is realized in several stages. The first stage consists of computing the variables depended on external data (i.e. the values of such variables is computed outside given MSCG). The set of points where variable U is calculated at the first stage is indicated as S^1_U. The second stage consists of computing the variables depended on both external data and the variables computed at the first stage. The set of points where the variable U is calculated at the second stage is indicated as S^2_U. This process will be continued till U values are computed in all the points of the initial domain. The sequence of U variable computation domains $S^1_U,\ldots.S^2_U$ will be the result of the process. Computation of U variable in all the points of S^i_U is carried out parallel but transition from the one domain to another is sequential. Similar domains systems are to be designed for all the variables included into MSCG.

Complicity of the sets S_u^j structures caused impossibility of writing the traversal of all the points from S_u^j domains in the form of effectively realized loops' systems.

On the other hand the following representation seemed acceptable: a set of points belonged to the domain can be set by an equation of hyperplane given below:

$$L_u(k): p_1^u i_1 + \ldots + p_n^u i_n + \Delta_n = k \qquad (3)$$

where p_i^u, Δ_u, k - integers.

We arrange sets $L_u(j)$ in the following way. Let $L_u(1)$ contains the points where the computation of U values depends on the external data only and these points satisfy a condition given below:

$$p_1^u i_1 + \ldots + p_n^u i_n + \Delta_n = 1$$

Set $L_u(2)$ consists of the points where the computation of U values depends on the external data and on the values computed in the points of set $L_u(1)$ besides these points satisfy a condition given below:

$$p_1^u i_1 + \ldots + p_n^u i_n + \Delta_n = 2$$

We'll continue this process till all the values of variable U in the initial domain are computed

Example: Let's consider relations:

$$X_{i,j} = f_1(X_{i-1,j+1}; Y_{i-1,j+2})$$

$$Y_{i,j} = f_2(X_{i+2,j-1}; Y_{i+3,j-1})$$

set on the domain $1 \le i \le 7, \quad 1 \le j \le 7$

The value in the (i,j)square points out number K of corresponding set S^k. E.g. the value in (4,4) square of X variable equal to 7 means that variable X in the point (4,) is computed on the 7th step of model's functioning.

S_x^k

1	1	1	1	1	1	1
1	2	2	2	2	2	2
1	7	6	4	4	4	3
1	8	8	7	6	5	5
1	10	9	9	8	7	6
1	12	11	10	10	9	8
1	14	13	12	11	11	10

7 6 5 4 3 2 1

1 2 3 4 5 6 7 $\longrightarrow i$

$\uparrow j$

6	5	3	3	3	1	1
7	7	5	5	4	1	1
9	8	7	6	6	1	1
11	10	9	8	7	1	1
13	12	11	10	9	1	1
14	13	13	12	11	1	1
1	1	1	1	1	1	1

S_y^k

Difficulties in the designing of loops' system for effective traversal of S^k set points are evident.

Let's create the system of sets belonging to the plane that is defined by the equations:

$$L_x(k): \quad -i+2j+4=k$$

$$L_y(k): \quad -i+2j+5=k$$

The tables where the values in the squares determine belonging of a particular point to the set are given below:

$L_x(k)$

7	6	5	4	3	2	1
9	8	7	6	5	4	3
11	10	9	8	7	6	5
13	12	11	10	9	8	7
15	14	13	12	11	10	9
17	16	15	14	13	12	11
19	18	17	16	15	14	13

$i \rightarrow$ 7 6 5 4 3 2 1

$\uparrow j$

8	7	6	5	4	3	2
10	9	8	7	6	5	4
12	11	10	9	8	7	6
14	13	12	11	10	9	8
16	15	14	13	12	11	10
18	17	16	15	14	13	12
20	19	18	17	16	15	14

1 2 3 4 5 6 7

$L_y(k)$

Differed from sets S^k sets $L(k)$ have regular structure. It gives a chance to write the sequence of computations in the form of not complicated loop structure.

Given tables show us that at each step of computation there is a set of points where variables values are computed simultaneously(parallel).

E.g. the values of variable Y in the points (1.7) (3,6) (5,5) (7,4) and the values of variable X in the points (2,6), (4.5), (6.7) are computed parallel at step K=8.

All the points where variable X is computed parallel lie on the one line.

The values of Y in the points (2,6) (4.5) (6,4) and the values of X in the points (1.6) (3,5) (5,4) (7,3) are computed at the next step. All the points where variable $Y(X)$ is computed at this step also lie on the one line.

Let's assume that there is such a hyperplane where we can orginize computations in the way described above. Introduce a definition.

Definition. Front of computation of variable U at the step K is a set of points $\{i_1,....i_n\}$ where the value of U may be computed parallel and the points belong to hyperplane $L_u(k): p_1^u i_1+...+p_n^u i_n+\Delta_n=k$

Changing K values we obtain the family of hyperplanes which covers all the points of the initial domain.

We can show that fronts of computations included into MSCG and satisfy condition $i_k \geq 0 \quad k = \overline{1,n}$ are parallel hyperplanes.

Consider the relation included into MSCG

$$U_{i_1,i_2,\ldots i_n} = f_u(\ldots; V_{i_1+a_1,\ldots i_n+a_n};\ldots)$$

point $(i_1,\ldots i_n) \in L_u(k_0)$, point $(i_{1+a_1},\ldots i_{n+a_n}) \in L_v(k_1)$

the value of V must be computed before U value is available, so

$$p_1 i_1 + \ldots + p_n i_n + \Delta_n = k_0 > k_1 = p_1(i_1 + a_1) + \ldots + p_n(i_n + a_n) + \Delta_v$$

The main inequality is derived from the one given above. All the parameters of hyperplane must satisfy it

$$\Delta_u - \Delta_v - \sum_{k=1}^{n} p_k a_k > 0 \qquad (5)$$

Consider computations' fronts relations for variables U and V included into MSCG.

$$L_u(k): \sum_l p_l i_l + \Delta_u = k$$

$$L_v(k): \sum_l p_l i_l + \Delta_v = k$$

Lets consider some point with coordinates $(i_1^0,\ldots i_n^0)$ belonged to the initial domain. The computation of U value in this point is carried out at K_u step and the computation of V value is done at $K_v$step, i.e.

$$\sum_l p_l i_l^0 + \Delta_u = K_u$$

$$\sum_l p_l i_l^0 + \Delta_v = K_v$$

Subtracting the second equation from the first one we obtain:

$$\Delta_u - \Delta_v = K_u - K_v$$

I.e. difference $\Delta_u - \Delta_v$ shows us "delay' ("outstripping") of Uvariable in comparison with V variable.

Now we can devide the question of loop process organization into two questions.

1. Definition of computation's front hyperplane's parameters.

2. Definition of coordinates of the points lying on the front's hyperplane (i.e. determination of the parameters of the loops providing traversal of all the hyperplane's points).

Let the system of relations (1) be set. Parameters $(p_1,\ldots,p_n)$ are to determine for variables $X^1,\ldots,X^m$. Besides values of Δ_i are to be defined for every variable X^i. System of inequality (5) defines the conditions for $p_1,\ldots,p_n,\Delta$: $i=\overline{1,m}$ to satisfy.

Computations in the points lying on the one hyperplane are done parallel. But transition from the one plane to another is sequential. Thus the number of the planes covering all the points of the initial domain determines (K_p) number of a sequential steps of computations.

The formula for K_p is given below.

$$K_p = \sum_{i=1}^{n} |p_i| * (N_i - M_i)$$

where N_i, M_i are the upper and the low boundaries of changing i coordinate domain.

Thus the task of wave's front determination comes to solution of LIP problem of the type given below.

Find $p_1,\ldots,p_n$, $\Delta_1,\ldots,\Delta_m$ which satisfy

$$\min \sum_{i=1}^{n} |p_i| * (N_i - M_i)$$

with following constraints

$$\Delta_l - \Delta_{l_1} - \sum_{i=1}^{n} p_i * \delta_{ll_1}^i > 0 \quad l, l_1 = \overline{1,m}$$

where $\delta_{ll_1}^i$ is a displacement to index i in the relation defined dependence of X^l on X^{l_1} during the computation.

Nowadays direct algorithm of integer programming is used for solving such a task [4].

Note 1. Further we'll consider only those MSCG where formulated task of LIP has a solution.

We don't consider the question of computation organization in the case of sequential computing. E.g. for the relations of the following type

$$A_{i,j} = f_1(A_{i-1,j+1}; B_{i,j})$$

$$B_{i,j} = f_2(A_{i,j-1}; B_{i+1,j-1})$$

with $2 \le i \le M \quad 2 \le j \le N$

there is no front of computations (as it is defined above). It is derived from the fact that inequalities of (5) type for the given example has the following form

$$p_1 - p_2 > 0$$

$$-\Delta_B > 0$$

$$\Delta_B + p_2 > 0$$

$$-p_1 + p_2 > 0$$

We can see from the given inequalities (the first and the last inequalities are conflicting) that there is no values p_1, p_2 defining the front of computations for variables A and B but the computations for the given relations may be organized in the form of the simple sequential loops.

Note 2. Presence of absolute values in the function demands solving of several tasks and choosing of the optimal one. We have to face the real tasks with the dimension of no more than 3 and it allows to judge of practical applications of this approach.

Program scheme for the one operator.

Let's assume that variable's values computed in the left part are not used in the right part i.e. there is no recurrent dependence. It means that all the values of the used variables are known and the computation can be done parallel for all the points of the domain. The scheme of operators' loop has the following form:

DO LABEL $i_1=M_1,N_1$

.....

DO LABEL $i_n=M_n,N_n$

LABEL $x(i_1,i_2,...,i_n)$=.....

Now assume that there is a recurrent dependence in the operator. In general the task of finding the parameters of wave's front is to be solved first and then you may start designing the loops' scheme. The exception might be such an index (or the group of indexes) which displacements has the one sight. If such an index exists we won't consider it. The header of the loop corresponding to particular index has the form:

DO LABEL $i_k=M_k,N_k$

in the case of the negative displacements

DO LABEL $i_k=M_k,N_k,-1$

in the case of the positive displacements.

Let's create a new graph Gn derived from the initial graph G by deleting the arcs where corresponding mark is strictly positive. Then use procedure of finding index with displacements of one sign. If there is no one like that then for the rest dependencies on other indexes you'll have to solve the task of determination of computations' front parameters.

Hyperplane of computations' front is said to be defined. The scheme of the program in this case will have the form:

DO LABEL $K=K_b,K_e$

DO LABEL CONC ALL $(i_1,...,i_n) \in L(k)$

LABEL < operator, variable computation>

where $L(k)$ is an equation of hyperplane. Record CONC ALL $(i_1,...,i_n) \in L(k)$ means that the loop provides processing of all the points lying on the hyperplane $L(k)$ and the computation in this points may be carried out parallel. External loop is sequential and it provides transition from the points of the one hyperplane to another

Front of computations has the form:

$$p_1 i_1 + ... + p_n i_n = k$$

(In the case of one relation $\Delta = 0$)

Changing K values we obtain the system of hyperplanes which covers all the points lying in the initial domain. It is evident that minimal and maximal values of K variable are on the domain's boundaries and they are equal to:

$$K_h = \sum_{p_i<0} p_i N_i + \sum_{p_i>0} p_i M_i \qquad (6)$$

$$K_c = \sum_{p_i<0} p_i M_i + \sum_{p_i>0} p_i N_i \qquad (7)$$

Now let's consider the question of designing loops' system providing processing of all the points of hyperplane. First let's assume that all $p_i \neq 0 \quad i = \overline{1.n}$ and the greatest common devisor of the numbers p_i are equal to 1.

Indicate $t_1 = p_2 i_2 + ... + p_n i_n \qquad (8)$

Initial equation assumes the form

$$p_1 i_1 + t_1 = k - \Delta \qquad (9)$$

Solution of this equation depends on the parameter d_1 (integer) and has the form:

$$i_1 = d_1$$

$$t_1 = (k - \Delta) - p_1 d_1 \qquad (10)$$

It is natural that you need to choose only those solutions which belong to the initial domain. E.g. value i_1 must satisfy inequalities

$$M_1 \le i_1 \le N_1$$

i.e. $M_1 \le d_1 \le N_1$. Besides let's define the boundaries of changing value of t_1.

Indicate

$$t_{1\max} = \sum_{\substack{i=2 \\ p_i>0}}^{n} p_i N_i + \sum_{\substack{i=2 \\ p_i<0}}^{n} p_i M_i \qquad (11)$$

$$t^1_{\min} = \sum_{\substack{i=2 \\ p_i<0}}^{n} p_i N_i + \sum_{\substack{i=2 \\ p_i>0}}^{n} p_i M_i \qquad (12)$$

It is evident that $t^1_{\min} \le t_1 \le t^1_{\max}$. Substituting instead of t_1 its expression into the last inequalities and solving it on d_1 we obtain

$$\frac{t^0_1 - t^1_{\min}}{p_1} \ge d_1 \ge \frac{t^0_1 - t^1_{\max}}{p_1} \qquad \text{if} \qquad p_1 > 0$$

and

$$\frac{t^0_1 - t^1_{\max}}{p_1} \ge d_1 \ge \frac{t^0_1 - t^1_{\min}}{p_1} \qquad \text{if} \qquad p_1 < 0$$

where $t^0_1 = k - \Delta$

In the result the domain of changing value of d_1 is set by inequalities

$$d^e_1 \ge d_1 \ge d^b_1$$

where $d^b_1 = \left\lceil \max(M_1, \frac{t^0_1 - t^1_{\max}}{p_1}) \right\rceil$

$$d^e_1 = \left\lfloor \min(N_1, \frac{t^0_1 - t^1_{\min}}{p_1}) \right\rfloor \qquad (13)$$

for $p_1 > 0$

$$d^b_1 = \left\lceil \max(M_1, \frac{t^0_1 - t^1_{\min}}{p_1}) \right\rceil$$

$$d^e_1 = \left\lfloor \min(N_1, \frac{t^0_1 - t^1_{\max}}{p_1}) \right\rfloor \qquad (14)$$

for $p_1 < 0$

Consider the rest part of the formula

$$p_2 i_2 + \ldots + p_n i_n = t^0_2$$

where $t^0_2 = (k - \Delta) - p_1 d_1 = t^0_1 - p_1 d_1$

Indicate

$$t_2 = p_3 i_3 + \ldots + p_n i_n$$

Equation (15) assume the form

$$p_2 i_2 + t_2 = t^0_2$$

Its solution has the form

$$i_2 = d_2$$

$$t_2 = t_2^0 - p_2 d_2$$

Let's define

$$t_{\max}^2 = \sum_{\substack{i=3 \\ p_i>0}}^{n} p_i N_i + \sum_{\substack{i=3 \\ p_i<0}}^{n} p_i M_i$$

$$t_{\min}^2 = \sum_{\substack{i=3 \\ p_i<0}}^{n} p_i N_i + \sum_{\substack{i=3 \\ p_i>0}}^{n} p_i M_i$$

From the inequalities

$$M_2 \le i_2 \le N_2$$

$$t_{\max}^2 \ge t_2 \ge t_{\min}^2$$

we obtain the range of changing the value

$$d_2^e \ge d_2 \ge d_2^b$$

where $d_2^b = \left\lceil \max(M_2, \frac{t_2^0 - t_{\max}^2}{p_2}) \right\rceil$

$$d_2^e = \left\lfloor \min(N_2, \frac{t_2^0 - t_{\min}^2}{p_2}) \right\rfloor \qquad (16)$$

for $p_2 > 0$. In the case of $p_2 < 0$ the formulae are derived in the same way.

We'll continue this process till the initial equation assumes the form

$$p_{n-1} i_{n-1} + p_n i_n = t_{n-1}^0 \qquad (17)$$

Let's consider that p_{n-1} and p_n are reciprocals. Then at first using Euclid's algorithm, we solve the equation

$$p_{n-1} i_{n-1} + p_n i_n = 1 \qquad (18)$$

Let $i_{n-1}^0 \ldots\ldots i_n^0$ be the solution of the equation (18). Then it will be evident that

$$i_{n-1} = i_{n-1}^0 * t_{n-1}^0 + d_{n-1} * p_n$$

$$i_n = i_n^0 * t_{n-1}^0 + d_{n-1} * p_{n-1}$$

are the solution of equation (17). From the conditions

$$M_{n-1} \le i_{n-1} \le N_{n-1}$$

$$M_n \le i_n \le N_n$$

we obtain the range of changing d_{n-1} value

$$d_{n-1}^e \ge d_{n-1} \ge d_{n-1}^b$$

where $d^b_{n-1} = \left\lceil \max\left(\frac{M_{n-1} - t_\alpha}{p_n}, \frac{t_\beta - N_n}{p_{n-1}}\right) \right\rceil$ (19)

$$d^e_{n-1} = \left\lfloor \min\left(\frac{N_{n-1} - t_\alpha}{p_n}, \frac{t_\beta - M_n}{p_{n-1}}\right) \right\rfloor \quad (20)$$

in the case of $p_{n-1}, p_n > 0$

where $t_\alpha = i^0_{n-1} * t^0_{n-1}$ $\qquad t_\beta = i^0_n * t^0_{n-1}$

The rest expressions for d^b_{n-1}, d^e_{n-1} values with the different combinations of the signs p_{n-1}, p_n values are derived in the same way. Thus the scheme of the loop provided processing of all the [points of hyperplane has the following form:

$t^0_1 = k - \Delta$

$D^b_1 = \ldots$

$D^e_1 = \ldots$

DO LABEL $D_1 = D^b_1, D^e_1$

$t^0_2 = t^0_1 - p_1 * D1$

$D^b_2 = \ldots$

$D^e_2 = \ldots$

DO LABEL $D_2 = D^b_2, D^e_2$

.....

$t_\alpha = i^0_{n-1} * t^0_{n-1}$

$t_\beta = i^0_n * t^0_{n-1}$

$D^b_{n-1} = \ldots$

$D^e_{n-1} = \ldots$

DO LABEL $D_{n-1} = D^b_{n-1}, D^e_{n-1}$

LABEL .....

Note: Assume that as some index i_k exists then parameter of hyperplane $p_k = 0$. Hence the equation has the form:

$$p_1 i_1 + \ldots + p_{k-1} i_{k-1} + p_{k+1} i_{k+1} + \ldots + p_n j_n + \Delta = k \quad (21)$$

In this case we consider algorithm of loop designing for the equation as it is given above(21). The header of the loop corresponding to the direction i_k will have a form:

DO LABEL $i_k = M_k, N_k$

Loop on the direction i_k can be formulated as both external and embedded.

Structure of index expressions.

The order of the embedded loops for processing of all the points of hyperplane are determined above. Each direction of the space corresponds to some loop index. For the first n-2 directions correspondence has the form:

$$i_k = d_k$$

For the last two indexes correspondence has more complicated form:

$$i_{n-1} = t_\alpha + d_{n-1} * p_n$$

$$i_n = t_\beta + d_{n-1} * p_{n-1}$$

I.e. index structure of the relation $(i_1, \ldots, i_n)$ corresponds to the following structure in the program:

$$(d_1, \ldots, d_{n-2}, t_\alpha + d_{n-1} * p_n, t_\beta - d_{n-1} * p_{n-1})$$

The numbers p_{n-1}, p_n must be reciprocals for the solution of equation(18). Let them be non reciprocals. Then before start an algorithm we must find two reciprocals among n numbers of $p_1, \ldots, p_n$. Such numbers must be found by all means because at the beginning we assume that the greatest common devisor of the numbers $p_1, \ldots, p_n$ are 1. This constraint doesn't change an algorithm. In this case only the correspondence of the space directions and loop indexes is changed. The correspondence for this case is given below.

$$i_1 = d_1$$

.....

$$i_{r-1} = d_{r-1}$$

$$i_{r+1} = d_r$$

.....

$$i_{s-1} = d_{s-2}$$

$$i_s = d_{s-1}$$

.....

$$i_n = d_{n-2}$$

$$i_r = t_\alpha + d_{n-1} * p_s$$

$$i_s = t_\beta + d_{n-1} * p_r$$

Front of computations with complicated parameters.

Previous computations were carried out having assumed that the greatest common devisor (δ) differs from 1. Let's consider changes in the algorithm in the case $\delta \neq 1$.

Let the initial front has the form:

$$L_1(k): \quad \delta(p_1 i_1 + \ldots + p_n i_n) + \Delta = k$$

Represent Δ in the form of $\Delta = \delta\tilde{\Delta} + r, \quad 0 \le r \le \delta$

Let's show that there are the points on the hyperplane for every value of K. Let the point with coordinates $(i_1^0, \ldots, i_n^0)$ lying on the hyperplane $L_1(k_0)$ be at the step k_0. Assume that the point on the hyperplane $L_1(k_0 + t)$ exists at the step $k_0 + t, \quad t < \delta$. Then

$$\delta(p_1 i_1^0 + \ldots + p_n i_n^0) + \Delta = k$$

$$\delta(p_1 i_1^1 + \ldots + p_n i_n^1) + \Delta = k_0 + t$$

are the equations where the following formula is derived from

$$p_1(i_1^1 - i_1^0) + \ldots + p_n(i_n^1 - i_n^0) = \frac{t}{\delta} \qquad (23)$$

There is an integer value in the equation from the left but there is none from the right. Hence the statement of the point existence on the hyperplane $L_1(k_0 + t)$ isn't right. Further more from the last equation we can see that the points for the initial equation of the hyperplane lie on the planes which are δ value apart, i.e. if there are the points on the plane $L_1(k_0)$, then the new points lie on the planes $L_1(k_0 + l\delta)$ where $l = \pm 1, \quad \pm 2, \ldots$

Proposition. The sequence of the computations for the front

$$L_2(k): \quad p_1 i_1 + \ldots + p_n i_n + \tilde{\Delta} = k \qquad (24)$$

is equivalent to the sequence of the computations for the initial front(22).

Proof. First let 's show that at the initial value $K = K_b^1$ for the initial front the computations are carried out in the same points and for the front in the form (24) for $K = K_b^2$ where

$$K_b^2 = \sum_{p>0} M_i p_i + \sum_{p<0} N_i p_i + \tilde{\Delta}$$

and $\tilde{\Delta}$ is taken from the representation $\Delta = \delta\tilde{\Delta} + r$

From the representation K_b^1 and K_b^2 there $K_b^1 = K_b^2 * \delta + r$ is derived

Consider point $(i_1^0, \ldots, i_n^0) \in L_1(K_b^1)$. Let's show that this point belongs to the front of (24) type for $K = K_b^2$ i.e. $(i_1^0, \ldots, i_n^0) \in L_2(K_b^2)$; $(i_1^0, \ldots, i_n^0) \in L_1(K_b^1)$,i.e.

$$\delta(p_1 i_1^0 + \ldots + p_n i_n^0) + \Delta = K_b^1 \qquad (25)$$

We must show that

$$p_1 i_1^0 + \ldots + p_n i_n^0 + \tilde{\Delta} = K_b^2$$

Substitute expression K_b^1 into (25)

$$\delta(p_1 i_1^0 + \ldots + p_n i_n^0) + \Delta = K_b^2 * \delta + r$$

Taking into consideration that $\Delta = \delta\tilde{\Delta} + r$ we have

$$\delta(p_1 i_1^0 + \ldots + p_n i_n^0 + \tilde{\Delta}) + r = K_b^2 * \delta$$

i.e. $\quad p_1 i_1^0 + \ldots + p_n i_n^0 + \tilde{\Delta} = K_b^2$

We have shown that the points lying on the front's plane exist not for every K for the initial equation. If the first are on plane $L_1(K_b)$ then the other lie on plane $L_1(K_b + \delta * m)$. Let's show that for any point lying on this plane there exists the plane of front set in the form (24) $\left(L_2(K_b^2 + m)\right)$

Let $(i_1^0, \ldots, i_n^0) \in L_1(K_b^1 + \delta * m)$

Let's show that given point

$$(i_1^0, \ldots, i_n^0) \in L_2(K_b^2 + m)$$

$$\delta(p_1 i_1^0 + \ldots + p_n i_n^0) + \tilde{\Delta} * \delta + r = K_b^1 + \delta m$$

We must show $\quad p_1 i_1^0 + \ldots + p_n i_n^0 + \tilde{\Delta} = K_b^2 + m$

We have $K_b^1 = K_b^2 * \delta + r$

Substituting the expression for K_b^1 we obtain

$$\delta(p_1 i_1^0 + \ldots + p_n i_n^0 + \tilde{\Delta}) + r = K_b^2 * \delta + r + \delta * m$$

$$p_1 i_1^0 + \ldots + p_n i_n^0 + \tilde{\Delta} = K_b^2 + m$$

It is derived from the proposition 1 that in the case $\delta \neq 1$ there is enough to transit from the front of (22) type to the front of (24) type and then to use the algorithm of loop parameters definition given above.

Loop structure for some relations in the case of the simple front.

Let the front for X^j variables included into MSCG has the form

$$L_j(k): \quad p_1 i_1 + \ldots + p_n i_n + \Delta_j = k \quad j = \overline{1,m}$$

The dimension of all the variables is said to be n and $p_1, \ldots, p_n$ are reciprocals. As appears from the above if the dimensions of the domains and Δ values are the same for the variables then loop headers are the same too. Thus in this case the formulae setting computations for such variables may be written together in the body of designed loops' system. If this condition isn't performed then first we define K_b and K_e as

$$K_b = \min_j K_b^j, \quad K_e = \max_j K_e^j \quad j = \overline{1,m}$$

Further we must put the test of the following type before every structure

$$K_b \le K \le K_e$$

and only when the condition is performed we'll start the loop providing search of the points on the front's plane with current K value for the given variable.

Note. Though if there are still variables set on the one domain and had the same Δ value they may be united.

Thus the scheme of loop in this case has the following form:

K_b=...

K_e=...

DO LABEL $K = K_b, K_e$

IF $K \ge K_b^j \wedge K \le K_e^j$

DO LABELj CONC ALL $(i_1,\ldots,i_n) \in L_j(K)$

$\overset{h}{x}(i_1,\ldots,i_n)$=.....

.....

LABELj $\overset{j_h}{x}(i_1,\ldots,i_n)$=.....

.....

IF $K \ge K_b^r \wedge K \le K_e^r$

DO LABELr CONC ALL $(i_1,\ldots,i_n) \in L_r(K)$

$\overset{h}{x}(i_1,\ldots,i_n)$=.....

.....

LABELr $\overset{r_h}{x}(i_1,\ldots,i_n)$=.....

.....

LABEL CONTINUE

where values of K_b^j, K_e^j variables are common for variables $\overset{h}{x},\ldots,\overset{j_h}{x}$ and K_b^r, K_e^r - for variables $\overset{r_1}{x},\ldots,\overset{r_h}{x}$.

Consider the case when $p_1,\ldots,p_n$ aren't reciprocals. Remember MSCG consists of the several operators.

Let the computations' front of U variable has the form

$$L_u(k): \delta(p_1 i_1 + \ldots + p_n i_n) + \tilde{\Delta}_u \delta + r_u = k$$

and for V

$$L_v(k): \delta(p_1 i_1 + \ldots + p_n i_n) + \tilde{\Delta}_v \delta + r_v = k$$

It is evident that computation of U and V are carried out for the same K values in the case when $r_u = r_v$.

It is not difficult to show that if $r_v > r_u$ then computation of V is carried out $r_v - r_u$ steps later after computation of U variable (though it can be done in the different points of the domain). While transition to the front of computations with simple parameters the computations of all the variables are carried out for the same k values. In this case the scheme of the program must be changed.

All the relations are united in the groups.

The relations included in the one group are carried out parallel as it is done before. Though every group begins processing only after the operators of the previous group end it.

Let's formulate the rules of uniting the relations into the one group and the order of the group movement. Uniting of the relations is done in the following way. Let's represent variable Δ_j in the form of $\tilde{\Delta}_j * \delta + r_j$. All the relations where r_j values are equal are united in the one group. Define the order of the groups' movement . Consider the initial front of the wave $(\delta(p_1 i_1 + \ldots + p_n i_n) + \Delta = k)$ and determine the minimal value of $K^j_{\min}$ for every $\overset{j}{x}$ variable. Let's arrange the groups according to the following rule. The first will be the group which includes the relation where $K_{\min}$ assume the value (on all the values of $K^j_{\min}$). Given group unites all the relations where $r_i = r_0$. The next group unites the relations where $r = r_0 + 1$. If there is no such group then choose the group with $r = r_0 + 2$ etc. Note $r_i < \delta$. If we define the place of the group with $r = \delta - 1$ then the next group will be with $r = 0$, etc.

Note. Let's consider the question if the formulae included into the different groups $(r_i \neq r_j)$ have the same minimal value $K_{\min}$ i.e. $K^i_{\min} = K^j_{\min}$.

Let the wave's fronts for $\overset{i}{x}$ and $\overset{j}{x}$ has the form:

$$L_i(k): \delta(p_1 i_1 + \ldots + p_n i_n + \tilde{\Delta}_i) + r_i = k$$

$$L_j(k): \delta(p_1 i_1 + \ldots + p_n i_n + \tilde{\Delta}_j) + r_j = k$$

Assume that $K^i_{\min} = K^j_{\min}$.

$$p_1(i^i_1 - i^j_1) + \ldots + p_n(i^i_n - i^j_n) + \tilde{\Delta}_i - \tilde{\Delta}_j = \frac{r_j - r_i}{\delta}$$

Because $r_i \neq r_j$ and $|r_i - r_j| < \delta$ our assumption $K^i_{\min} = K^j_{\min}$ isn't right.

The scheme of the program is given below.

K_b=...

K_e=...

DO 1 $K = K_b, K_e$

Computations on the relations included into the 1st group

Computations on the relations included into the 2nd group

.....

Computations on the relations included into the $\delta - 1$ group

1 CONTINUE

Transition from the computations of the one group to another is sequential. Computations of the relations included into the one group are organized in the way similar to the case of the front with simple parameters i.e. parallel inside each relation and besides each relation separately.

Examples.

The examples illustrating given method of loop designing are considered in this part.

Example 1. The domain of changing indexes is a rectangular $i = 3..M; \quad j = 3..N$

The following relations are set

$$A_{i,j} = B_{i,j-1}$$

$$B_{i,j} = A_{i-2,j+2}; B_{i,j-1}$$

The LIP program has the following form:

find p_1, p_2 and Δ_B where functional $Z = M * |p_1| + N * |p_2|$ assumes minimum with the constraints

$$p_2 - \Delta_B > 0$$

$$\Delta_B - 2p_2 + 2p_1 > 0$$

$$p_2 > 0$$

The solution is $p_1 = 2, \quad p_2 = 1 \quad \Delta_B = 0$

Program loop is given below.

$K_b = 9$

$K_e = 2 * M + N$

DO 1 $K = K_b, K_e$

DB=MAXX(3,(K-N)/2)

DE=MINN(M,(K-3)/2)

DO 2 *D=DB,DE*

$A(D, K - 2 * D) = B(D, K - 2 * D - 1)$

2 $B(D,K-2*D)=A(D-2,K-2*D+2);B(D,K-2*D-1)$

1 CONTINUE

Note. Function $MAXX(X,Y)$ gives us the nearest integer exceeded the greatest of the numbers X and Y as a result. Function $MINN(X,Y)$ gives us the nearest integer which doesn't exceed the least of the numbers X and Y.

Embedded loop may be carried out for all D values . This loop corresponds to the computation of the values A and B on particular line $(2*i+j=K)$ with fixed K value . Transition of the lines (loop on K) is sequential. Common number of the steps in the loop is $2M+N-8$.

If we don't solve the LIP task we may first analyse index displacements on each direction. Note that displacements on index i has one sign. Hence we may not take into consideration link between the variables on this index in those cases when the displacement is non-zero. We can see the result : variable B doesn't depend on A. We can propose the following scheme of the loop.

DO 1 $i=3,M$

DO 2 $j=3,N$

$A(i,j)=B(i,j-1)$

1 $B(i,j)=A(i-2,j+2);B(i,j-1)$

Program loop seems easier in this case. First you mustn't compute the boundaries of changing indexes of the embedded loops (as you must in the case of the first solution). Second the structure of the index expressions is simple enough. Though loop designed in this way is sequential (both on i and j). You may do parallel only computations of A and B when i and j are fixed. Common number of sequential steps is $(M-2)*(N-2)$ in this case.

Example 2. Let the relations be set

$$X_{i,j}=f_x(X_{i+1,j-1};Y_{i+1,j+1};Z_{i+1,j})$$

$$Y_{i,j}=f_y(X_{i,j-1};Y_{i+1,j};Z_{i-1,j-1})$$

$$Z_{i,j}=f_z(X_{i-1,j-1};Y_{i-1,j};Z_{i,j-1})$$

the domain of changing indexes

$$i=2,\ldots,M$$

$$j=2,\ldots,N$$

The fronts of computations for X,Y,Z are indicated $L_x(k)$, $L_y(k)$, $L_z(k)$ correspondingly. LIP problem solution has the form:

$$p_1=-2,\quad p_2=6,\quad \Delta_x=5,\quad \Delta_y=0,\quad \Delta_z=3$$

The values of p1 and p2 are not reciprocals. The fronts of computations has the form:

$$L_x(k):\quad -i+3j+2=k\quad r_x=1$$

$L_y(k)$: $-i+3j=k \quad r_y=0$

$L_z(k)$: $-i+3j+1=k \quad r_z=1$

It follows from given above that computations of values X and Z are carried out parallel (because X and Z are included in the one group and Y in another one) and sequentially after the computation of Y.

The scheme of the program is given below.

$K_b=-M+6$

$K_e=-2+3*N$

DO LABEL $K=K_b,K_e$

DB_y=$MAXX((2+K)/3,2)$

DE_y=$MINN((M+K)/3,2)$

DO LABEL1 $DY=DB_y,DE_y$

LABEL1 $Y(3*DY-K,DY)=X(3*DY-K,DY-1);Y(3*DY-K+1,DY);$

$Z(3*DY-K-1,DY-1)$

DB_x=$MAXX(K/3,2)$

DE_x=$MINN((M+K-2)/3,N)$

DO LABEL2 $DX=DB_x,DE_x$

LABEL2 $X(3*DX-K+2,DX)=X(3*DX-K+3,DX-1);Y(3*DX-K+3,DX+1);$

$Z(3*DX-K+3,DX)$

DB_z=$MAXX((K+1)/3,2)$

DE_z=$MINN((M+K-1)/3,N)$

DO LABEL3 $DZ=DB_z,DE_z$

LABEL3 $Z(3*DZ-K+1,DZ)=X(3*DZ-K,DZ-1);Y(3*DZ-K,DZ);$

$Z(3*DZ-K+1,DZ-1)$

LABEL CONTINUE

Every turn of the loop on K consists of two sequential steps. At the first step the values of variable Y in the front's point $-i+3j=k$ are computed parallel, at the second step the values of variable X in the front's points $i-+3j+2=k$ and the values of variable Z in the front's points-$i+3j+1=k$. The order of computations corresponding components of matrixes X, Y, Z in the case M=7, N=7,. E.G. $X(5,5)$=27 means that the value of variable X in the point (5,5) is computed at 27 step (the first computation is carried out at step 1).

X Y Z

44	43	41	39	37	35	7
39	37	35	33	31	29	6
33	31	29	27	25	23	5
27	25	23	21	19	17	4
21	19	17	15	13	11	3
15	13	11	9	7	5	2
2	3	4	5	6	7	

$i \rightarrow$

$\uparrow j$

40	38	36	34	32	30	7
34	32	30	28	26	24	6
28	26	24	22	20	18	5
22	20	18	16	14	12	4
16	14	12	10	8	6	3
10	8	6	4	2	1	2

43	42	39	37	35	33	7
37	35	33	31	29	27	6
31	29	27	25	23	21	5
25	23	21	19	17	15	4
19	17	15	13	11	9	3
13	11	9	7	5	3	2

Bibliography.

1. Andrianov A.N., Efimkin K.N., Zadykhailo I.B., Podderugina N.B. " The NORMA language". Preprint KIAM AS USSR, 1985, N165.

2. Zadykhailo I.B., Pimenov S.P. " Semantics of the NORMA language". Preprint KIAM AS USSR, 1986, 139

3. Ranegold E., Nivergelt U.,Deo N. " Combinatoric algorithm. Theory and practice" m., Mir, 1980.

4. Who T. " integer programming and streams in nets" m., mir, 1974.

Russian Academy of Science
Keldysh Institute of Applied Mathematics

Andrianov A.N., Andrianova E.A.

Organization of loop process on nonprocedural specification.

Moscow 1996

Introduction.

NORMA is a programming language [1] aimed at automation of mathematical physics problems solutions on parallel computer systems.

The NORMA language allows elimination of a programming phase in transition from formulae specified by a technical expert to a program itself. There is no much difference between formulae and NORMA specifications. In fact these formulae are input data for a translator.

Synthesis of output program is carried out automatically during the translation from NORMA. The order and the way of performing calculations (parallel, vector or sequential) is determined automatically. The order of the language's sentences is arbitrary (information dependencies are revealed and taken into account during the organization of computing process). There are no such programming terms as memory, loop, control operators in the language. Output program is generated with the architecture of a target computer as a guide.

In fact the program in NORMA is a nonprocedural specification of the problem to be solved. The synthesis of output program raises some mathematical problems but they are solvable in the case of NORMA language.

Some Norma peculiarities makes the process of automatic object program design available for practice realization. They are:

1. Index expressions of calculated variables has the form $i \pm c$

where i - index name, c - integer constant.

NORMA is a language with single assignment. Any value can be assigned to a variable only once (only once to each point of domain - to the variables defined on domain).The first constraint defines the class of formulae which can be used for the problem's solution . It isn't strict in practice as the index expressions of other type are very rare.

Memory allocation and the problems of its economy caused the second constraint. These problems can be solved at the translation stage. The second constraint simplifies the problem of output program synthesis.

The problem of output program is to be solved during the translation. Solving this problem is based on the analysis of the graph of information dependencies. The Most Strongly Connected subGraphs (MSCG) are chosen from the graph. In general case organization of computations for the nodes requires use of special methods.

Assume is a principal operator in NORMA. This operator sets the relations between variables being calculated on a domain. Researching on the subject of computational process organization has been doing for a long time. The notion of local relation on an array was introduced in [5] and called parameter record. Synthesis of loop programs for parameter record in the case of one-dimension array was investigated in [6]. The problems of existence and degree of parallelism of computational process on multidimensional arrays are considered in [7]. Organization of loop computations for relations set on half limited domains is described in [8]. The problems of parallel program synthesis on the base of special formalization of application domain are investigated in [9].

The purpose of this paper is to specify the method of designing loop operators which realizes the relations included into MSCG. Given work is a continuation of [2].

1 FRONT OF COMPUTATIONS.

Executive part in the NORMA language may be represented in the following form:

$$\begin{aligned} \overset{1}{X}_I &= f_1(\overset{1}{X}_{I+D_{11}},\ldots,\overset{m}{X}_{I+D_{1m}}) \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \overset{m}{X}_I &= f_m(\overset{1}{X}_{I+D_{m1}},\ldots,\overset{m}{X}_{I+D_{mm}}) \end{aligned} \qquad (1)$$

Here $I=\{i_1,\ldots,i_n\}$ denotes the array of indexes corresponded to calculated variables $\overset{1}{X},\ldots,\overset{m}{X}$, $D_{i,j}=\{d^1_{i,j},\ldots,d^n_{i,j}\}$ denotes the array of displacements to index variables, $d^k_{i,j}$ - integer constants in NORMA as it is defined in Norma.

Note. The use of one and the same value with different index expressions is possible in the right part of the relations. Besides the independence from some variables is also acceptable. The form of relation's system is used for simplification.

The computations defined by t relation are said to be performed in the domain:

$$M^t_k \le i_k \le N^t_k, \quad k=\overline{1,n} \qquad (2)$$

The values required for computations and lain outside the domain are considered known. The order of computations isn't set in an explicit way thus information dependencies are to be analysed and then computations are arranged on the result of analysis.

Directed graph is constructed in accordance with the given system of relations. Every relation corresponds to the graph's node. Information dependence of the following type:

$$\overset{k}{X}_I = f_k(\ldots,\overset{t}{X}_{I+D_{kt}},\ldots)$$

corresponds to the graph's arc from node $\overset{t}{X}$ to node $\overset{k}{X}$ with mark D_{kt}.

The most strongly connected subgraphs are searched in the obtained graph and the graph is reduced. A new graph is a non-loop and thus the procedure of arranging nodes according to the relation of "computed before" order is possible. Determination of the loop process for the operators not included into MSCG is easy.

The problem of loop process determination for the operators included into MSCG will be considered below.

At first we consider the methods of computations sequence informally.

Definition 1. Set of the points satisfied equation

$$\Lambda(\overset{t}{X},k)=\{i_1,\ldots,1_n\}:\ p_1i_1+\ldots+p_ni_n+\Delta^t_1=k, \qquad (3)$$

are called basically front and equation (3) - basical equation of $\overset{t}{X}$ variable at k step computations' front.

Note. Difference $\Delta_1^t - \Delta_1^l \quad t, l = \overline{1, m}$ defines several steps, that computations of variable $\overset{t}{X}$ "delay" (outstrip") from the computations of variable $\overset{l}{X}$ in the same point.

Note that the range of changing step k value is defined by all means for the initial domain (2). Let the range of step k changing has the form: $[k^{min}, k^{max}]$. Computation of the variable's values consists of the sequence of steps in the initial domain. At first value k^{min} is assigned to variable k : $k = k^{min}$ and the values of variable $\overset{t}{X}$ are computed in all the points of basical front (3). Then start with the next step. 1 is added to k value ($k = k+1$) and the value of $\overset{t}{X}$ is calculated on a new front. Given process is continued until k value assumes its maximum k^{max}.

Note. Every variable $\overset{l}{X}$ (when value of (k) step is fixed) is computed on the plane generally parallel to the plane where the value of $\overset{t}{X}$ is computed.

Definition 2. Basical equation where the Greatest Common Devisor (GCD) ($p_1,...,p_n$) = 1 is called *reduced basical equation.*

Let's consider the following system of equations:

$$\begin{aligned}
\Lambda_1(\overset{t}{X}, k_1): p_{1,1}i_1 + \ldots + p_{1,n}i_n + \Delta_1^t &= k_1 \\
\Lambda_2(\overset{t}{X}, k_2): p_{2,1}i_1 + \ldots + p_{2,n}i_n + \Delta_2^t &= k_2 \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\
\Lambda_r(\overset{t}{X}, k_r): p_{r,1}i_1 + \ldots + p_{r,n}i_n + \Delta_r^t &= k_r
\end{aligned} \qquad (4)$$

Here $p_{s,j}, \Delta_s^t \quad j = \overline{1, n} \quad s = \overline{1, r} \quad t = \overline{1, m}$ are integer numbers.

The first equation from system (4) is basical.

Definition 3. *Front of computations of* $\overset{t}{X}$ *value at* k_1 step is a set of the points satisfied basical equation, the order of the points' traversal is specified by the equations(4).

The order of variables' computations is considered below. The values of variable $\overset{t}{X}$ in the opines with integer coordinates belonged to the plane set by the basical equation of the front are to be calculated when k_1 (from range $[k_1^{min}, k_1^{max}]$) is fixed. The order of other points' traversal is specified by other equations. The second equation defines the family of n-dimensional hyperplanes (according to k_2 values changing from k_2^{min} to k_2^{max}). By adding the first equation we obtain the family of n-1-dimensional hyperplanes which cover n-dimensional hyperplane set by the basical equation. Thus the points on the n-1-dimensional hyperplane are calculated first (when $k_2 = k_2^{min}$). Then 1 is added to k_2 value and the values in the points of the next n-1-dimensional plane are calculated etc. Let's consider

fixed n-1-dimensional hyperplane (defined by the first two equations when $k_1 = \tilde{k}_1$ and $k_2 = \tilde{k}_2$). The order of the traversal of the points belonged to this hyperplane is determined in the same way: the third equation is considered and varying k_3 values from k_3^{min} to k_3^{max} we obtain the family of hyperplanes covering all the points of n-1-dimensional hyperplane. The third equation with the equations defined n-1- dimensional hyperplane sets n-2 -dimensional hyperplane (when $k_3 = \tilde{k}_3$ is fixed).If there are no other equations then the order of the traversal of the points on this hyperplane is arbitrary (i.e. parallel). In the other case the next equation is considered etc. This procedure will be continue until the last equation is considered. It is evident that if $r=n$ then the traversal of the points of the initial domain will be sequential i.e. point by point. If $r<n$ the computation may be carried out parallel in the points n- r+1-dimensional hyperplane. In particular if the front of computation consists of the one (basical) equation then the computations of the given variable may be carried out independently in all the points of the plane.

Note. The questions connected with the determination of the front of computations are described in [3-4].

2 Computations on the basical front

Let's consider the problem of the order's determination of the initial domain's points' traversal when the front of computation is set by the basical equation:

$$\Lambda(\overset{l}{X},k) = \{i_1,...,i_n\} : p_1 i_1 + ... + p_n i_n + \Delta_1^l = k \qquad (5)$$

Denote $\sigma_1 = GCD(p_1,...,p_n)$. Thus every coefficient of the equation p_i, $i = \overline{1,n}$ is expressed in the following way: $p_i = \sigma_1 * \tilde{p}_i$. Represent equation (5) in the form:

$$\sigma_1 * (\tilde{p}_1 i_1 + ... + \tilde{p}_n i_n) + \Delta_1^l = k \qquad (6)$$

Denote $\tilde{p}_1 i_1 + ... + \tilde{p}_n i_n = t_1$. Then equation (6) assumes the form:

$$\sigma_1 t_1 - k = -\Delta_1^l$$

Write the solution of this equation:

$$\begin{aligned} t_1 &= d_1^l \\ k^l &= \Delta_1^l + \sigma_1 d_1^l \end{aligned} \qquad (7)$$

Parameter d_1^l is used for a sequential examination of all the planes covering the initial domain. As the parameter has been obtained for every variable then let's use the one which is common for all the variables. From $k^l = k^t$, $(l,t = \overline{1,m})$ follows:

$$d_1^l = d_1^t + \frac{\Delta_1^t - \Delta_1^l}{\sigma_1}$$

Let $$\tilde{d}_1^{k_1} = \max_{\overline{1,m}} \{d_1^l\}$$

Denote $R^1_{lk_1} = \dfrac{\Delta^{k_1}_1 - \Delta^l_1}{\sigma_1}$ and represent $R^1_{lk_1}$ in the form

$$R^1_{lk_1} = [R^1_{lk_1}] + \{R^1_{lk_1}\}$$

where $[R^1_{lk_1}]$ - integer part of a number and $\{R^1_{lk_1}\}$ - its fractional part.

Express all $d^l_1 \quad (l = \overline{1,m})$ through $\widetilde{d}^{k_1}_1$:

$$d^l_1 = \widetilde{d}^{k_1}_1 - [R^1_{lk_1}] \qquad (8)$$

Parameter d^l_1 can assume only integer values. Sequence of k step's initial value must be kept in the transition to the common parameter. Thus variable r^l_1 is to be linked with every $\overset{l}{X}$:

$$r^l_1 = \{R^1_{lk_1}\}$$

Now one and the same parameter $\widetilde{d}^{k_1}_1$ is used for the sequential examination of all the planes. When $\widetilde{d}^{k_1}_1$ is fixed the computation of variables are arranged on increasing of r^l_1 values.

Define the range of $\widetilde{d}^{k_1}_1$ values changing for every $\overset{l}{X}$:

$$\begin{aligned} d^{\min}_{l,1} &= \sum_{\overline{p}_i>0} \overline{p}_i M^l_i + \sum_{\overline{p}_i<0} \overline{p}_i N^l_i + [R^1_{lk_1}] \\ d^{\max}_{l,1} &= \sum_{\overline{p}_i>0} \overline{p}_i N^l_i + \sum_{\overline{p}_i<0} \overline{p}_i M^l_i + [R^1_{lk_1}] \end{aligned} \qquad (9)$$

Common range of $\widetilde{d}^{k_1}_1$ parameter's changing is a segment $[d^{\min}_1, d^{\max}_1]$ where

$$d^{\min}_1 = \min_{l=1,m}\{d^{\min}_{l,1}\}, \quad d^{\max}_1 = \max_{l=1,m}\{d^{\max}_{l,1}\}$$

The traversal of the initial domain's points is carried out by the sequential examination of the planes. Transition from the examination of the one plane to another is performed in the loop changing values of loop parameter $\widetilde{d}^{k_1}_1$ from $d^{\min}_1$ to $d^{\max}_1$. Range $(d^{\min}_1, d^{\max}_1)$ is common for all the variables. As every variable $\overset{l}{X}$ has its own range of computation steps $(d^{\min}_{l,1}, d^{\max}_{l,1})$ the test of the type given below is placed before the computation of the variable at $\widetilde{d}^{k_1}_1$ step:

IF $\widetilde{d}^{k_1}_1 \in (d^{\min}_{l,1}, d^{\max}_{l,1})$ THEN ...

Let's consider the question connected with the determination of the points with integer coordinates on the plane set by the following equation:

$$\widetilde{p}_1 i_1 + \ldots + \widetilde{p}_n i_n + [R^1_{lk_1}] = \widetilde{d}^{k_1}_1 \qquad (10)$$

Definition 4. Further the transition from the basical equation (5) to the reduced basical equation (10) will be called *algorithm of reduction.*

Represent equation (10) in the following form:

$$\sigma_2(p_1^1 i_1 + \ldots + p_{n-1}^1 i_{n-1}) + \overline{p}_n i_n = \widetilde{d}_1^{k_1} - [R_{lk_1}^1] \qquad (11)$$

where $\sigma_2 = GCD(\widetilde{p}_1, \ldots, \widetilde{p}_{n-1})$. Denote $t_2 = p_1^1 i_1 + \ldots + p_{n-1}^1 i_{n-1}$. In this case the equation has the following solution;

$$\begin{aligned} t_2 &= (\widetilde{d}_1^{k_1} - [R_{lk_1}^1])t_2^0 \pm \overline{p}_n d_2 \\ i_n &= (\widetilde{d}_1^{k_1} - [R_{lk_1}^1])i_n^0 \pm \sigma_2 d_2, \end{aligned}$$

where t_2^0 and i_n^0 are the solution of $\sigma_2 t_2 + \overline{p}_n i_n = 1$ equation.

Note. Choice of the sign is arbitrary in this case. The domain of changing of the values of t_2 variable is determined by the following range: $t_2 \in (t_2^{\min}, t_2^{\max})$ where

$$\begin{aligned} t_2^{\min} &= \sum_{\substack{p_i^1 > 0 \\ i=1,n-1}} p_i^1 M_i^t + \sum_{\substack{p_i^1 < 0 \\ i=1,n-1}} p_i^1 N_i^t, \\ t_2^{\max} &= \sum_{\substack{p_i^1 > 0 \\ i=1,n-1}} p_i^1 N_i^t + \sum_{\substack{p_i^1 < 0 \\ i=1,n-1}} p_i^1 M_i^t \end{aligned}$$

Taking into consideration the domain of t_2 variable changing and the boundaries of index variable i_n changing the range of parameter d_2 changing can be defined.

Then we consider the following equation:

$$p_1^1 i_1 + \ldots + p_{n-1}^1 i_{n-1} = (\widetilde{d}_1^{k_1} - [R_{lk_1}^1])t_2^0 \pm p_n d_2 \qquad (12)$$

Denote the right part by T_1. Let $\sigma_3 = GCD(p_1^1, \ldots, p_{n-2}^1)$. Represent equation (12) in the form:

$$\sigma 3(\widetilde{p}_1^1 i_1 + \ldots + \widetilde{p}_{n-2}^1 i_{n-2}) + p_{n-1}^1 i_{n-1} = T_1,$$

and denoting $t3 = \widetilde{p}_1^1 i_1 + \ldots + \widetilde{p}_{n-2}^1 i_{n-2}$ write the solution of the last equation:

$$\begin{aligned} t_3 &= T_1 t_3^0 \pm p_{n-1}^1 d_3 \\ i_{n-1} &= T_1 i_{n-1}^0 \mp \sigma_3 d_3 \end{aligned}$$

Variables t_3^0 and i_{n-1}^0 are the solution of the equation $\sigma_3 t_3 + p_{n-1}^1 i_{n-1} = 1$. The range of d_3 parameter changing is determined in the same way as for parameter d_2.

Then the process of solving will continue till the following equation is obtained:

$$p_1^{n-2} i_1 + p_2^{n-2} i_2 = T_{n-2}$$

Its solution has the form:

$$\begin{aligned} i_1 &= T_{n-2} i_1^0 \pm p_2^{n-2} d_n \\ i_2 &= T_{n-2} i_2^0 \mp p_1^{n-2} d_n \end{aligned}$$

Thus the traversal's scheme of the initial domain's points (2) when the from of computation (5) is set has the following form:

$d_1^{\min} = \ldots$

$d_1^{\max} = \ldots$

DO 1 $d_1 = d_1^{\min}, d_1^{\max}$

$d_2^{\min} = \ldots$

$d_2^{\max} = \ldots$

DO 1 $d_2 = d_2^{\min}, d_2^{\max}$

.....

$d_n^{\min} = \ldots$

$d_n^{\max} = \ldots$

DO 1 $d_n = d_n^{\min}, d_n^{\max}$

.....

IF $d_1 \in (d_{l,1}^{\min}, d_{l,1}^{\max}) \& \ldots \& d_n \in (d_{l,n}^{\min}, d_{l,n}^{\max})$ THEN

C The expressions of the index variables are omitted.

$\overset{l}{X}(\ldots) = \ldots$

.....

IF $d_1 \in (d_{l,1}^{\min}, d_{l,1}^{\max}) \& \ldots \& d_n \in (d_{l,n}^{\min}, d_{l,n}^{\max})$ THEN

$\overset{l}{X}(\ldots) = \ldots$

.....

1 CONTINUE

Note 1. Sequence of the formulae is determined based on the parameters r_1^l values.

Note 2. The case when the values of some coefficient $p_k = 0$ corresponds to a loop along the direction i_k with initial value M_k and the final value N_k.

3 Computations on the front of common type

An algorithm of loop parameters' determination for the fronts of computations set by the system of equations (4) is considered. At first we prove a theorem . We'll need its result in the further work.

Theorem 1. There exists matrix $\mathbf{A}_n(p_1, \ldots, p_n)$ with integer elements (dimension $n \times n$). The first row of the matrix and its $\det \mathbf{A}_n(p_1, \ldots, p_n) = 1$ consist of the coefficients of the basical equation.

Proof. The proof will be performed by induction. For n=2 matrix has the form:

$$\mathbf{A}_2(p_1, p_2) = \begin{pmatrix} p_1 & p_2 \\ \beta & \alpha \end{pmatrix}$$

where α and β are the solution of $p_1\alpha - p_2\beta = 1$ equation.

Let's assume that everything is right for $n < k$. Prove it for $n = k$. Represent the resulting matrix in the form of two matrixes' multiplication: $\mathbf{A}_n = \mathbf{B}_n \cdot \mathbf{C}_n$ where matrix $\mathbf{B}_n$ has the form:

$$\mathbf{B}_n = \begin{pmatrix} a & 0 & \dots & 0 & p_n \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & 0 \\ \alpha & 0 & \dots & 0 & \beta \end{pmatrix}$$

where $a = GCD(p_1, \dots p_{n-1})$ and α, β are the solution of equation:

$$a\beta - p_n\alpha = 1$$

Note that $\det \mathbf{B}_n = a\beta - p_n\alpha = 1$.

Matrix $\mathbf{C}_n$ has the following form:

$$\mathbf{C}_n = \begin{pmatrix} & & & 0 \\ & \mathbf{A}_{n-1}(\varphi_1, \dots, \varphi_{n-1}) & & \vdots \\ & & & 0 \\ 0 & \dots & 0 & 1 \end{pmatrix}$$

where $\varphi_i = \dfrac{p_i}{a}, \quad i = \overline{1, n-1}.$

Matrix $\mathbf{A}_{n-1}(\varphi_1, \dots, \varphi_{n-1})$ exists by the induction' assumption.

$$\det \mathbf{C}_n = \det \mathbf{A}_{n-1}(\varphi_1, \dots, \varphi_{n-1}) = 1$$

Hence

$$\det \mathbf{A}_n = \det \mathbf{B}_n = \det \mathbf{C}_n = 1$$

Matrix $\mathbf{A}_n$ is integer as matrixes $\mathbf{B}_n$ and $\mathbf{C}_n$ are integer. Let's consider the first line of the matrix $\mathbf{A}_n(p_1, \dots, p_n)$. The elements of the first line are calculated on the formula:

$$a_{1,i} = \sum_k b_{1,k} c_{k,i} = a\varphi_i = p_i$$
$$a_{1,n} = p_n.$$

Now let's start considering the method of loop parameters' determination.

Front of computations for $\overset{l}{X} \quad l = \overline{1, m}$ variables is considered.

$$\begin{aligned} p_{1,1}i_1+\ldots,+p_{1,n}i_n+\Delta_1^l &= K_1 \\ \ldots\ldots\ldots\ldots\ldots & \\ p_{r,1}i_1+\ldots,+p_{r,n}i_n+\Delta_r^l &= K_r \end{aligned} \qquad (13)$$

Denote $\sigma_1 = NOD(p_{1,1},\ldots p_{1,n})$. Using the algorithm of reduction in **2** we transit to the following equations:

$$\begin{aligned} p_{1,1}i_1+\ldots,+p_{1,n}i_n+[R^1_{lk_1}] &= \tilde{a}_1^{k_1} \\ \ldots\ldots\ldots\ldots\ldots & \\ p_{r,1}i_1+\ldots,+p_{r,n}i_n+\Delta_r^l &= K_r \end{aligned} \qquad (14)$$

where $[R^1_{lk_1}]=\dfrac{\Delta_1^{k_1}-\Delta_1^l}{\sigma_1}$ and $\tilde{p}_{i,j}=\dfrac{p_{i,j}}{\sigma_1}$. Perform the following coordinate's transformation:

$$\mathbf{A}_n(p_{1,1},\ldots,p_{1,n})\cdot\begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix}=\begin{pmatrix} i_1^n \\ \vdots \\ i_n^n \end{pmatrix}$$

Matrix $\mathbf{A}_n$ is chosen according to theorem 1, where the first equation from (14) is chosen according to theorem 1 where the first equation from (14) is a reduced basical equation and the first row in matrix $\mathbf{A}_n$. Denote:

$$(p^n_{k,1},\ldots,p^n_{k,n})=(p_{k,1},\ldots,p_{k,n})\mathbf{A}_n^{-1} \quad k=\overline{2,r}$$

We'll consider the following system of equations instead of system of equations (13):

$$\begin{aligned} p^n_{2,2}i^n_2+\ldots,+p^n_{2,n}i^n_n+p^n_{2,1}(\tilde{a}_1^{k_1}-[R^1_{lk_1}])+\Delta_2^l &= K_2 \\ \ldots\ldots\ldots\ldots\ldots & \\ p^n_{r,2}i^n_2+\ldots,+p^n_{r,n}i^n_n+p^n_{r,1}(\tilde{a}_1^{k_1}-[R^1_{lk_1}])+\Delta_r^l &= K_r \end{aligned} \qquad (15)$$

where we use the fact that $i_1^n=\tilde{p}_{1,1}i_1+\ldots+\tilde{p}_{1,j}i_n=\tilde{a}_1^{k_1}-[R^1_{lk_1}]$. Using algorithm of reduction we transit from the system of equations (15) to the system of equations:

$$\begin{aligned} \tilde{p}^n_{2,2}i^n_2+\ldots,+\tilde{p}^n_{2,n}i^n_n+[R^2_{lk_2}] &= \tilde{a}_2^{k_2} \\ p^n_{3,2}i^n_2+\ldots,+p^n_{3,n}i^n_n+p^n_{3,1}(\tilde{a}_1^{k_1}-[R^1_{lk_1}])+\Delta_3^l &= K_3 \\ \ldots\ldots\ldots\ldots\ldots & \quad K_r \\ p^n_{r,2}i^n_2+\ldots,+p^n_{r,n}i^n_n+p^n_{r,1}(\tilde{a}_1^{k_1}-[R^1_{lk_1}])+\Delta_r^l &= K_r \end{aligned} \qquad (16)$$

where the expression for $R^2_{lk_2}$ has the form:

$$R^2_{lk_2}=\frac{\Delta_2^{k_2}-\Delta_2^l+p^n_{2,1}([R^1_{lk_1}]-[R^1_{k_2k_1}]}{\sigma_2}$$

Thus we have the system of equations similar to (14) differed only in smaller number of equations and index variables. Further we'll continue this process till we get the equation of the following type:

$$\tilde{p}^{n-r+2}_{r,r} i^{n-r+2}_r + \ldots + \tilde{p}^{n-r+2}_{r,n} i^{n-r+2}_n + \left[R^r_{lk_r}\right] = d^{k_r} \qquad r < n$$

$$\mathrm{sign}\{\tilde{p}^n_{n,n}\} i^n_n + \left[R^n_{lk_n}\right] = d^{k_n}_n \qquad r = n$$

The methods of derivation of index expressions for such an equation is given in **2**. We obtain the values for indexes: $i^{n-r+2}_r, \ldots, i^{n-r+2}_n$ (for the case $r < n$) as a result of the last equation's solution. Now we carry out the following sequence of back transformations for the initial indexes' derivation:

$$\begin{pmatrix} i^{n-r+3}_r \\ \vdots \\ i^{n-r+3}_n \end{pmatrix} = A^{-1}_{n-r+2} \begin{pmatrix} i^{n-r+2}_r \\ \vdots \\ i^{n-r+2}_n \end{pmatrix}$$

Further we carry out the following transformation using the fact that $i^{n-r+3}_{r-1} = d^{k_r-1}_{r-1} - \left[R^{r-1}_{l,k_r-1}\right]$:

$$\begin{pmatrix} i^{n-r+4}_{r-1} \\ \vdots \\ i^{n-r+4}_n \end{pmatrix} = A^{-1}_{n-r+3} \begin{pmatrix} d^{k_r-1}_{r-1} - \left[R^{r-1}_{l,k_r-1}\right] \\ \vdots \\ i^{n-r+3}_n \end{pmatrix}$$

Thus process will be continued till the values for the initial indexes of the task are obtained:

$$\begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} = A^{-1}_n \begin{pmatrix} d^{k_1}_1 - \left[R^1_{l,k_1}\right] \\ \vdots \\ i^n_n \end{pmatrix}.$$

4 The fronts of computations in 3-dimensional domain

To illustrate method of loop parameters determination given above the fronts for the variables defined on 3-dimensional domain are considered. The front of computations set by two equations are considered in the first part . The front of computations is set by three equations in the second part.

4.1 Sequential-parallel computations

Let's assume that the front for some variable is set by two following equations:

$$\begin{aligned} p_{1,1} i_1 + p_{1,2} i_2 + p_{1,3} i_3 + \Delta^l_1 &= K_1 \\ p_{2,1} i_1 + p_{2,2} i_2 + p_{2,3} i_3 + \Delta^l_2 &= K_2 \end{aligned} \qquad (17)$$

Basical equation defines the points of the plane where requested value of the variable is to be calculated. The second equation specifies the order of the points' traversal on the plane set by the first equation.

Informally it means the following things. The traversal of the points with integer coordinates belonged to the plane set by the first equation is to be organized. The order of the traversal is following. As the values of the points' coordinate must satisfy both equations (17) (that is the setting of the line in the space if K_2 and K_1 values are fixed) then really we need to organise the traversal of the points with integer coordinates along the set line. The computations of variables' values on such a

line may be carried out parallel. Our purpose is to find such expressions for indexes' values (i_1, i_2, i_3) (which are the expressions of some parameters d_1, d_2, d_3) that the value of K_2 variable increases when the values of parameters d_1, d_2, d_3 increase. Then increasing the value of K_2 variable we start the traversal of the points along the next line etc.

Denote $\sigma_1 = GCD(p_{1,1}, p_{1,2}, p_{1,3})$. Using algorithm of reduction transit from the system (17) to the following system of equations:

$$\begin{aligned} \tilde{p}_{1,1}i_1 + \tilde{p}_{1,2}i_2 + \tilde{p}_{1,3}i_3 + \left[R^1_{lk}\right] &= K_1 \\ p_{2,1}i_1 + p_{2,2}i_2 + p_{2,3}i_3 + \Delta^l_2 &= K_2 \end{aligned} \qquad (18)$$

Denote: $a = GCD(\tilde{p}_{1,1}, \tilde{p}_{1,2})$. Let's assume that $\tilde{p}_{1,1} = a\varphi_1$ and $\tilde{p}_{1,2} = a\varphi_2$. Note that $GCD(\varphi_1, \varphi_2) = 1$ and $GCD(a, \tilde{p}_{1,3}) = 1$. Denote by σ and γ the solutions of the equation: $\varphi_1\sigma - \varphi_2\gamma = 1$ and by β and α solutions of equation:

$a\beta - \tilde{p}_{1,3}\alpha = 1$.

Consider the following matrix:

$$\mathbf{A}_3 = \begin{pmatrix} a\varphi_1 & a\varphi_2 & \tilde{p}_{1,3} \\ \gamma & \sigma & 0 \\ \alpha\varphi_1 & \alpha\varphi_2 & \beta \end{pmatrix}$$

Transform the initial coordinates:

$$\mathbf{A}_3(a\varphi_1, a\varphi_1, \tilde{p}_{1,3}) \begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} = \begin{pmatrix} i^3_1 \\ i^3_3 \\ i^3_3 \end{pmatrix} \qquad (19)$$

and denote $(p^3_{2,1}, p^3_{2,2}, p^3_{2,3}) = (p_{2,1}, p_{2,2}, p_{2,3}) \cdot \mathbf{A}_3^{-1}$. The second equation assumes the form:

$$p^3_{2,1}i^3_1 + p^3_{2,2}i^3_2 + p^3_{2,3}i^3_3 + \Delta^l_2 = K_2$$

As $i^3_1 = \tilde{d}_1^{k_1} - \left[R^1_{lk_1}\right]$ then the second equation from (18) assumes the form:

$$\sigma_2(\tilde{p}^3_{2,2}i^3_2 + \tilde{p}^3_{2,3}i^3_3) - K_2 = -p^3_{2,1}(\tilde{d}_1^{k_1} - \left[R^1_{lk_1}\right] - \Delta^l_2 \qquad (20)$$

where $\sigma_2 = GCD(p^3_{2,2}, p^3_{2,3})$. Rewrite equation (20) in the form:

$$\sigma_2 t_1 - K_2 = -p^3_{2,1}(\tilde{d}^{k_1} - \left[R^1_{lk_1}\right] - \Delta^l_2 \qquad (21)$$

where $t_1 = \tilde{p}^3_{2,2}i^3_2 + \tilde{p}^3_{2,3}i^3_3$. The solution of this equation has the form:

$$\begin{aligned} t_1 &= d^l_2 \\ K^l_2 &= p^3_{2,1}(\tilde{d}_1^{k_1} - \left[R^1_{lk_1}\right] + \Delta^l_2 + \sigma_2 d^l_2 \end{aligned}$$

Then all the parameters d_2^l will be reduced to the common parameter $\tilde{d}_2^{k_2}$ and we start solving problem:

$$p_{2,2}^3 i_2^3 + p_{2,3}^3 i_3^3 + \left[R_{l k_2}^2\right] = \tilde{d}_2^{k_2}$$

Write the solution of this equation:

$$\begin{aligned} i_2^3 &= (\tilde{d}_2^{k_2} - \left[R_{l k_2}^2\right] i_2^0 \pm p_{2,3}^3 d_3^l \\ i_3^3 &= (\tilde{d}_2^{k_2} - \left[R_{l k_2}^2\right] i_3^0 \pm p_{2,2}^3 d_3^l \end{aligned} \qquad (22)$$

Note. The choice of the sign before parameter d_3^l is arbitrary. It corresponds to the fact that the traversal of the points is done by the sequential examination of the lines but the traversal along every line is done parallel on every point.

The explanations of some notations' usage is given below. In this case i_2^0 and i_3^0 are the solutions of Euclid's equation: $p_{2,2}^3 i_2^0 + p_{2,3}^3 i_3^0 = 1$. The variable $R_{l k_2}^2$ has the form:

$$R_{l k_2}^2 = \frac{p_{2,1}^3 (\left[R_{l k}^1\right] - \left[R_{k_2 k_1}^1\right]) + \Delta_2^{k_2} - \Delta_2^l}{\sigma_2}$$

The final solution is evident:

$$\begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} = A_3^{-1} \begin{pmatrix} d_1^{k_1} - \left[R_{l k_1}^1\right] \\ (\tilde{d}_2^{k_2} - \left[R_{l k_2}^2\right] i_2^0 \pm p_{2,3}^3 d_3^l \\ (\tilde{d}_2^{k_2} - \left[R_{l k_2}^2\right] i_3^0 \pm p_{2,2}^3 d_3^l \end{pmatrix}$$

3.2 Sequential computations

Let the front of some variables be set by three following equations:

$$\begin{aligned} p_{1,1} i_1 + p_{1,2} i_2 + p_{1,3} i_3 + \Delta_1^l &= K_1 \\ p_{2,1} i_1 + p_{2,2} i_2 + p_{2,3} i_3 + \Delta_2^l &= K_2 \\ p_{3,1} i_1 + p_{3,2} i_2 + p_{3,3} i_3 + \Delta_3^l &= K_3 \end{aligned} \qquad (23)$$

The front of computations set in this form determines strict sequential order of computation of the variables on the plane set by the basical equation. The first two equations set the family of the lines (when K_2 varies from $K_2^{\min}$ to $K_2^{\max}$) as it was in the previous case. At first the values of the variables in the points of the first line (when $K_2 = K_2^{\min}$) then increasing the value of K_2 by 1 we start the traversal of the points on the next line etc. The traversal of the points along every line was arbitrary in the previous case. When the front of computations is set by three equations the last one determines the direction of the traversal of the points along the lines.

As it was in the previous case instead of system of equations (23) we start considering the following ones:

$$\begin{aligned} \widetilde{p}_{1,1}i_1 + \widetilde{p}_{1,2}i_2 + \widetilde{p}_{1,3}i_3 + \left[R_{lk_1}\right] &= \widetilde{d}_1^{k_1} \\ p_{2,1}i_1 + p_{2,2}i_2 + p_{2,3}i_3 + \Delta_2^l &= K_2 \\ p_{3,1}i_1 + p_{3,2}i_2 + p_{3,3}i_3 + \Delta_3^l &= K_3 \end{aligned} \qquad (24)$$

Using the transformation of the coordinates (19) we obtain instead of the second and the third equations the following ones:

$$\begin{aligned} p_{2,2}^3 i_2^3 + p_{2,3}^3 i_3^3 + p_{2,1}^3(\widetilde{d}_1^{k_1} - \left[R_{lk_1}\right] + \Delta_2^l &= K_2 \\ p_{3,2}^3 i_2^3 + p_{3,3}^3 i_3^3 + p_{3,1}^3(\widetilde{d}_1^{k_1} - \left[R_{lk_1}\right] + \Delta_3^l &= K_3 \end{aligned} \qquad (25)$$

Use the algorithm of reduction and obtain the following system of equations:

$$\begin{aligned} \widetilde{p}_{2,2}^3 i_2^3 + \widetilde{p}_{2,3}^3 i_3^3 + \left[R_{lk_2}^2\right] &= \widetilde{d}_2^{k_2} \\ p_{3,2}^3 i_2^3 + p_{3,3}^3 i_3^3 + p_{3,1}^3(\widetilde{d}_1^{k_1} - \left[R_{lk_1}\right] + \Delta_3^l &= K_3 \end{aligned} \qquad (26)$$

Then using the transformation of the coordinates:

$$\mathbf{A}_2 \cdot \begin{pmatrix} i_2^3 \\ i_3^3 \end{pmatrix} = \begin{pmatrix} \widetilde{p}_{2,2}^3 & \widetilde{p}_{2,3}^3 \\ \beta & \alpha \end{pmatrix} \cdot \begin{pmatrix} i_2^3 \\ i_3^3 \end{pmatrix} = \begin{pmatrix} i_2^2 \\ i_3^2 \end{pmatrix},$$

where β and α are the solutions of equation $\widetilde{p}_{2,2}^3\alpha - \widetilde{p}_{2,3}^3\beta = 1$ and denoting $(p_{3,2}^2, p_{3,3}^2) = (p_{3,2}^3, p_{3,3}^3) \cdot \mathbf{A}_2^{-1}$ we obtain the second equation from (26) in the following form:

$$p_{3,2}^2 i_2^2 + p_{3,3}^2 i_3^2 + p_{3,1}^3(\widetilde{d}_1^{k_1} - \left[R_{lk_1}^1\right] + \Delta_3^l = K_3$$

As $i_2^2 = \widetilde{d}_2^{k_2} - \left[R_{lk_2}^2\right]$ we may rewrite the last equation in the following form:

$$p_{3,3}^2 i_3^2 - K_3 = -p_{3,1}^3(\widetilde{d}_1^{k_1} - \left[R_{lk_1}^1\right] - p_{3,2}^2(\widetilde{d}_1^{k_2} - \left[R_{lk_2}^2\right] - \Delta_3^l$$

The solution of this equation has the form:

$$\begin{aligned} i_3^2 &= \operatorname{sign}\{p_{3,3}^2\} d_3^l \\ K_2^l &= p_{3,1}^3(\widetilde{d}_1^{k_1} - \left[R_{lk_1}^1\right] + p_{3,2}^2(\widetilde{d}_1^{k_2} - \left[R_{lk_2}^2\right] + \Delta_3^l + \left|p_{3,3}^2\right| d_3^l \end{aligned}$$

Reduced to common parameter $d_3^{k_3}$ we obtain the solution for i_3^2 in the form:

$$i_3^2 = \operatorname{sign}\{p_{3,3}^2\}(\widetilde{d}_3^{k_3} - \left[R_{lk_3}^3\right]$$

Now we start the procedure of the initial indexes' derivation. At first we transform:

$$\begin{pmatrix} i_2^3 \\ i_3^3 \end{pmatrix} = \mathbf{A}_2^{-1} \begin{pmatrix} \widetilde{d}_2^{k_2} - \left[R_{lk_2}^2\right] \\ \operatorname{sign}\{p_{3,3}^2\}(\widetilde{d}_3^{k_3} - \left[R_{lk_3}^3\right] \end{pmatrix}$$

and then:

$$\begin{pmatrix} i_1 \\ i_2 \\ i_3 \end{pmatrix} = \mathbf{A}_3^{-1} \begin{pmatrix} d_1^{k_1} - \left[R_{lk_1}^1\right] \\ i_2^3 \\ i_3^3 \end{pmatrix}.$$

THE EXAMPLE OF LOOP OPERATORS' CONSTRUCTION

We consider the example of loop operator's construction for the following system of relations:

$$
\begin{aligned}
X_{i,j,k} &= f_x(X_{i,j-1,k}, Y_{i,j,k}) \\
Y_{i,j,k} &= f_y(X_{i-1,j,k}, Y_{i,j+1,k}, U_{i-1,j,k}, Z_{i,j,k}) \\
Z_{i,j,k} &= f_z(Y_{i,j,k+1}, Z_{i,j-1,k}) \\
U_{i,j,k} &= f_u(Y_{i,j,k+1}, U_{i,j,k+1}, V_{i-1,j-1,k}) \\
V_{i,j,k} &= f_v(U_{i,j+1,k}, V_{i,j,k-1})
\end{aligned}
$$

Set values are to be calculated in the domain set by the system of inequalities:

$$i \in (1,10); \quad j \in (1,10); \quad k \in (1,10).$$

For X, Y, Z, U and V variables the following equations of the computations' fronts are obtained:

$$
\begin{array}{llllll}
X: & 2i+1 = K_1, & j = K_2 & & & \\
Y: & 2i = K_1, & -2k+1 = K_2, & -j = K_3 & & \\
Z: & 2i = K_1, & -2k = K_2, & j = K_3 & & \\
U: & 2i+1 = K_1, & -k = K_2 & & & \\
V: & 2i+2 = K_1, & k = K_2 & & &
\end{array}
$$

Using the algorithm of reduction we obtain the following systems of computations' fronts.

$$
\begin{array}{llllll}
X: & i = K_1, & j = K_2 & & & \\
Y: & i = K_1, & -2k+1 = K_2, & -j = K_3 & & \\
Z: & i = K_1, & -2k = K_2, & j = K_3 & & \\
U: & i = K_1, & -k = K_2 & & & \\
V: & i+1 = K_1, & k = K_2 & & &
\end{array}
$$

Here the values of parameters r_1^l, $l = X, Y, Z, U, V$ has the following form:

$$r_1^X = 1, \quad r_1^Y = 0, \quad r_1^Z = 0, \quad r_1^U = 1, \quad r_1^V = 0.$$

It is easy to prove that the range of parameter d_1 changing is (1,11). Besides the range for every value (except V value) is (1,10) and for V - (2,11).

DO 1 $d_1 = 1,11$

IF $d_1 \in (1,10)$ IF $d_1 \in (2,11)$

$\boxed{Y \quad Z}$ $\boxed{V}$

C ---

C $X \quad U$

C $Y, Z, V,$

C $r_1^X = r_1^U = 1$

IF $d_1 \in (1,10)$ IF $d_1 \in (1,10)$

$\boxed{X}$ $\boxed{U}$

1 CONTINUE

Determination of loop operator's parameters for X, U and V values isn't difficult. The fronts of computations of this variables are different thus they are considered separately. The procedure of reduction to the common parameter isn't necessary. Thus we'll consider one of these variables.

The front of computations for variable X has the following form:

$$\begin{aligned} i &= d_1 \\ j &= K_2 \end{aligned}$$

Matrix of transformation in this case is a unit one. Change the system of notations:

$\mathbf{A}_3$

$$\mathbf{A}_3 \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} \tilde{i} \\ \tilde{j} \\ \tilde{k} \end{pmatrix}$$

Corresponds to (20) the second equation assumes the form:

$$\tilde{j} = d_2.$$

The form of the index expressions follows from (22):

$$\begin{aligned} \tilde{j} &= d_2 \\ \tilde{k} &= \pm d_3 \end{aligned}$$

Transiting to the initial notations:

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \mathbf{A}_3^{-1} \cdot \begin{pmatrix} d_1 \\ d_2 \\ \mp d_3 \end{pmatrix}$$

We obtain the solution :

$$\tilde{j} = d_2.$$

$$\tilde{j} = d_2$$
$$\tilde{k} = \pm d_3$$

Thus the scheme of X variable computation has the following form:

DO 2 $d_2 = 1,10$

DO 2 $d_3 = 1,10$

$X(d_1, d_2, d_3) = \ldots$

2 CONTINUE

Nested loop (on d_3) may be carried out parallel. It is evident because before parameter d_3 stands sign "$\mp$".

Consider the fronts of computations for Y and Z variables:

$$Y: \; i = K_1, \quad -2k+1 = K_2 \quad -j = K_3$$
$$Z: \; i = K_1, \quad -2k = K_2, \quad j = K_3$$

As the front of computations is set by three equations then both these variables are calculated strictly sequential. Unit matrix of transformation and equations (25) has the following form:

$$Y: \quad -2\tilde{k}+1 = K_2 \quad -\tilde{j} = K_3$$
$$Z: \quad -2\tilde{k} = K_2, \quad \tilde{j} = K_3$$

Using the algorithm of reduction we obtain the following fronts of computations:

$$Y: \; -\tilde{k} = d_2 \quad -\tilde{j} = K_3$$
$$Z: \; -\tilde{k} = d_2 \quad \tilde{j} = K_3$$

Parameter r_2 has the following values:

$$r_2^Y = 1, \quad r_2^Z = 0.$$

Matrix of transformation for obtained fronts of computations has the following form:

$$\mathbf{A}_2 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

Transform the coordinates:

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \tilde{j} \\ \tilde{k} \end{pmatrix} = \begin{pmatrix} \hat{j} \\ \hat{k} \end{pmatrix}$$

One equation for the front of computations of Y variable is left as a result:

$$-\hat{k} = K_3$$

and one equation for the front of computation of Z value:

$$\hat{k} = K_3$$

Then we obtain the solution in the initial notations. At first we carry out the transformation of the following type:

$$\begin{pmatrix} \tilde{j} \\ \tilde{k} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} d_2 \\ -d_3 \end{pmatrix} = \begin{pmatrix} -d_3 \\ -d_2 \end{pmatrix}$$

And finally we obtain:

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \mathbf{A}_3 \cdot \begin{pmatrix} d_1 \\ -d_3 \\ -d_2 \end{pmatrix}$$

The solution for Z variable is done in the same way.

The scheme of the program is given below.

DO 1 $d_1 = 1,11$

IF $d_1 \in (1,10)$ THEN

C Z

C Z

C Y $r_2^Y = 1$

DO 2 $d_2 = -1,01$

DO 3 $d_3 = 1,10$

3 $Z(d_1, d_3, -d_2) = fz(Y(d_1, d_3, -d_2 + 1), Z(d_1, d_3 - 1, -d_2))$

C Y

DO 4 $d_3 = -1,0,-1$

$Y(d_1, -d_3, -d_2) = f_Y(X(d_1 - 1, -d_3, -d_2), Y(d_1, -d_3 + 1, -d_2), U(d_1 - 1, -d_3, -d_2), Z(d_1, -d_3, -d_2))$

4 CONTINUE

2 CONTINUE

C V

IF $d_1 \in (2,11)$ THEN

DO 5 $d_2 = 1,10$

C d_3

DO 5 $d_3 = 1,10$

5 $V(d_1 - 1, d_3, d_2) = f_V(U(d_1 - 1, d_3 + 1, d_2), V(d_1 - 1, d_3, d_2 - 1))$

C X

IF $d_1 \in (1,10)$ THEN

DO 6 $d_2 = 1,10$

C d_3

DO 6 $d_3 = 1,10$

6 $X(d_1, d_3, d_2) = f_X(X(d_1, d_2 - 1, d_3), Y(d_1, d_2, d_3))$

C U

IF $d_1 \in (1,10)$ THEN

DO 7 $d_2 = -10,-1$

C d_3

DO 7 $d_3 = -10,-1$

$U(d_1,-d_3,-d_2) = f_U(Y(d_1,-d_3,-d_2+1), U(d_1,-d_3,-d_2+1), V(d_1-1,-d_3-1,-d_2))$

7 CONTINUE

1 CONTINUE

PostScript

Special thanks to I.B. Zadykhailo and K.N. Efimkin for their attention to the work and useful notes.